

Budapesti Műszaki és Gazdaságtudományi Egyetem

Elektronikus Eszközök Tanszéke

Pohl László

A programozás alapjai

Budapest, 2010

© Pohl László, 2010

Minden jog fenntartva

Az elektronikus könyv, abban a formában, ahogy van, szabadon terjeszthető. Részletei csak abban az esetben jelentethetők meg az eredeti könyvtől külön, ha az idézett részletről egyértelműen kiderül, hogy idézet, és megfelelő formában szerepel hivatkozás erre a műre.

Tartalom

I. RÉSZ – ALAPOZÁS.....	8
1. A programozás.....	8
1.1 A számítógép felépítése.....	8
1.2 A programozás alapjai.....	8
2. Az algoritmusok megadása.....	10
Teafőző robot	10
Gondolkodjunk együtt!.....	12
3. Egyszerű algoritmusok és kódolásuk C nyelven	12
1. feladat:.....	12
2. feladat	13
3. feladat	14
Második megoldás.....	15
Gondolkodjunk együtt!.....	16
Oldja meg egyedül!	17
Az integrált fejlesztőkörnyezet.....	17
Program létrehozása	17
Fordítás, futtatás, hibakeresés	18
4. Hogy működik a fordító?	20
5. A C programok felépítése	20
5.1 A szintaxis leírása	20
5.2 Egy egyszerű C program elemei.....	21
6. Típusok, változók, konstansok, műveletek.....	24
6.1 Másodfokú egyenlet gyökei.....	24
6.2 Átlag	25
6.3 Fordítva.....	26
Gondolkozzunk együtt!.....	27
Oldja meg egyedül!	27
7. A C utasításai	28
Gondolkozzunk együtt!.....	32
Oldja meg egyedül!	32
8. Egyszerű függvények	32

<i>Gondolkozzunk együtt!</i>	34
<i>Oldja meg egyedül!</i>	34
II. RÉSZ: ISMERKEDÉS A C NYELVVEL	35
9. Tömbök.....	35
<i>Gondolkozzunk együtt!</i>	36
<i>Oldja meg egyedül!</i>	36
10. Számábrázolás és az egyszerű adattípusok	37
10.1. Számábrázolás.....	37
10.2. C alaptípusok	38
<i>Gondolkozzunk együtt!</i>	39
<i>Oldja meg egyedül!</i>	39
11. Bitműveletek.....	39
<i>Gondolkozzunk együtt!</i>	40
<i>Oldja meg egyedül!</i>	40
12. Bemenet és kimenet, ASCII.....	41
12.1 A karakterek ASCII kódja.....	41
12.2 <i>getchar/putchar, gets/puts, fgets</i>	42
12.3 <i>printf</i> és <i>scanf</i>	43
<i>Gondolkozzunk együtt!</i>	44
<i>Oldja meg egyedül!</i>	44
13. A számítógépek felépítéséről és működéséről.....	44
14. Összetett és származtatott adattípusok, típusdefiníció	47
14.1 Összetett és származtatott típusok	47
14.2 Saját típus létrehozása – a típusdefiníció	48
<i>Gondolkozzunk együtt!</i>	48
<i>Oldja meg egyedül!</i>	48
15. Operátorok	49
15.1 Logikai értékek és egész számok	49
15.2 A C nyelv operátorai	49
<i>Gondolkozzunk együtt!</i>	52
<i>Oldja meg egyedül!</i>	52
16. Pointerek	53
16.1 A számítógép memóriája.....	53
16.2 A pointer	53
16.3 Pointeraritmetika.....	54

<i>Gondolkozzunk együtt!</i>	54
<i>Oldja meg egyedül!</i>	54
17. Érték és cím szerinti paraméterátadás; tömbök, pointerek és függvények.....	55
17.1 Paraméterátadás.....	55
17.2 Tömbök átadása függvénynek	56
17.3 Tömb és pointer újra.....	57
17.4 Pointertömb.....	57
<i>Gondolkozzunk együtt!</i>	58
<i>Oldja meg egyedül!</i>	58
18. Dinamikus memóriakezelés.....	58
18.1 sizeof.....	59
18.2 Többdimenziós dinamikus tömb.....	59
18.3 A lefoglalt memória bővítése.....	60
<i>Gondolkozzunk együtt!</i>	61
<i>Oldja meg egyedül!</i>	61
19. Sztringek	61
19.1 Könyvtári sztringfüggvények	61
19.2 Algoritmusok	63
<i>Gondolkozzunk együtt!</i>	64
<i>Oldja meg egyedül!</i>	64
20. A programozás menete	64
20.1 Programstruktúra tervezése.....	65
20.2 Algoritmus megadása.....	65
21. Összetettebb C programok kellékei	66
21.1 Több modulból álló programok	66
21.2 A C programok általános felépítése	67
21.3 Láthatóság, tárolási osztályok.....	67
<i>Gondolkozzunk együtt!</i>	68
<i>Oldja meg egyedül!</i>	68
22. A parancssor	69
<i>Gondolkozzunk együtt!</i>	69
<i>Oldja meg egyedül!</i>	69
23. Fájlkezelés.....	70
23.1 Bináris fájlok	70
23.2 Szöveges fájlok	71

23.3 Szöveges fájlok és bináris fájlok összevetése.....	73
23.4 Szabványos bemenet és kimenet.....	73
Gondolkozzunk együtt!.....	73
Oldja meg egyedül!.....	73
III. RÉSZ: HALADÓ PROGRAMOZÁS	74
24. Dinamikus önhivatkozó adatszerkezetek 1. – láncolt listák	74
24.1 Egyszerű láncolt lista	74
14.2 Láncolt lista strázsával, függvényekkel, beszúrással	75
24.3 Két irányban láncolt lista	76
24.3 Fésűs lista.....	77
Gondolkozzunk együtt!.....	77
Oldja meg egyedül!.....	77
25. Állapotgép (véges automata)	78
25.1 Kommentyszűrő.....	78
Gondolkozzunk együtt!.....	79
Oldja meg egyedül!.....	79
26. Rekurzió.....	79
26.1 A verem adatszerkezet	79
26.2 Függvényhívás	80
26.3 Rekurzió	81
Gondolkozzunk együtt!.....	82
Oldja meg egyedül!.....	82
27. Dinamikus önhivatkozó adatszerkezetek 2. – fák.....	83
27.1 Bináris fa	83
27.2 Rendezett fa – rendezőfa, kiegyensúlyozottság és kiegyensúlyozatlanság.....	84
27.3 A fabejárás típusai	85
27.4 Több ágú fák	85
27.5 Két példa	85
Gondolkozzunk együtt!.....	86
Oldja meg egyedül!.....	86
28. Függvénypointer	86
Gondolkozzunk együtt!.....	87
Oldja meg egyedül!.....	87
29. Keresés tömbben.....	88
Gondolkozzunk együtt!.....	89

<i>Oldja meg egyedül!</i>	89
30. Rendezés.....	89
30.1 Buborék algoritmus	89
30.2 Közvetlen kiválasztásos rendezés	90
30.3 Süllyesztő rendezés	91
30.4 Közvetlen beszűrő rendezés.....	91
30.5 Shell rendezés	92
30.6 Gyorsrendezés	92
30.7 Szabványos könyvtári gyorsrendező függvény: a qsort.....	93
30.8 Láncolt lista rendezése gyorsrendező algoritmussal	93
<i>Gondolkozzunk együtt!</i>	93
<i>Oldja meg egyedül!</i>	93
31. Preprocesszor	94
<i>Gondolkozzunk együtt!</i>	95
<i>Oldja meg egyedül!</i>	95
32. Bitmezők, union.....	96
32.1 Struktúra bitmezői	96
32.2 Union	96
<i>Gondolkozzunk együtt!</i>	97
<i>Oldja meg egyedül!</i>	97
33. Változó paraméterlistájú függvények.....	97
<i>Gondolkozzunk együtt!</i>	98
<i>Oldja meg egyedül!</i>	98
34. Hasznos függvények a C-ben	98
<i>Gondolkozzunk együtt!</i>	99
<i>Oldja meg egyedül!</i>	99
35. Programozási érdekességek	99
35.1 Programozás több szálon	99
Tárgymutató	102

I. RÉSZ – ALAPOZÁS

1. A programozás

1.1 A számítógép felépítése



A számítógépek fő részegységei a processzor, a memória és a perifériák. A **memória** általános tároló, mely utasításokat és adatokat tartalmaz. A **processzor** beolvassa a memóriából az utasításokat és az adatokat, az utasítások alapján műveleteket végez, a műveletek eredményét pedig visszairja a memóriába, valamint vezérli a **perifériákat**: adatokat küld számukra és adatokat olvas be onnan.

A **memória** ► olyan, mint egy nagyon hosszú táblázat, melynek minden cellájában egy szám van, ami csak 0 vagy 1 lehet:

0	1	1	1	0	0	1	0	0	0	1	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Azért csak 0 és 1, mert a számítógépek **kettes számrendszert** használnak, és a kettes számrendszerben csak ez a két számjegy van. Azért kettes számrendszert használnak, és nem pl. tízest, mert így egy kapcsoló elég a szám tárolásához: ha a kapcsoló be van kapcsolva, az 1, ha ki van kapcsolva, az 0. Tízest számrendszerben tíz különböző állapotot kellene megkülönböztetni, ami sokkal nehezebb, mint kettőt. Sokkal egyszerűbb megállapítani, hogy egy lámpa világít-e vagy sem, mint azt, hogy milyen erősen világít egy tízfokozatú skálán.

A kettes (azaz **bináris**) számrendszer számjegyeit angolul binary digitnek, röviden **bit**nek nevezzük.

Ahhoz, hogy a memóriát hatékonyan lehessen használni, a benne tárolt információkhoz tetszőleges sorrendben hozzá kell, hogy tudjunk férni (a másik lehetőség a sorban egymás után lenni). Az információk tetszőleges sorrendű elérése a memóriában (Random Access Memory – RAM) úgy valósítható meg legegyszerűbben, ha minden bitről tudjuk, hanyadik a memória kezdetétől számítva, és ez alapján hivatkozunk rá.

Egy bit nagyon kevés információt tárol, ezért a számítógépek tervezői úgy döntöttek, hogy nem engedik meg minden bit címzését, hanem csak bitsoportokét, amit **bájt**nak nevezünk, mert így egyszerűbb az áramkörök felépítése. A számítógépekben tehát úgy választjuk ki a nekünk kellő adatot vagy utasítást, hogy megmondjuk, hanyadik bájt a memória kezdetétől számítva. Ezt a sorszámot **memóriacím**nek nevezzük.

Egy bájt méretét úgy választották meg, hogy egy karakter (azaz betű, számjegy, írásjegy, stb.) beleférjen. Sokféle bájt méretet használtak: 6, 7, 8, 9 bites bájtok is léteztek. Ezek közül manapság a 8 bites bájt szinte egyeduralgódó. A könyvben ismertett C nyelv szabványa előírja, hogy csak olyan rendszerben használható a C nyelv, ahol egy bájt legalább 8 bites, azonban futnak C programok pl. 9 bites bájtokat használó rendszerekben is, ezt később se feledjük!

Egy nyolcbites bájt a következő számokat tárolhatja: 00000000=0, 00000001=1, 00000010=2, 00000011=3, 00000100=4, 00000101=5, ..., 11111111=255. Azaz összesen 256 féle értékünk van (2^8), 0-tól 255-ig. A bájtunkat azonban értelmezhetjük másképp is, például ha azt mondjuk, hogy az első bit nem számjegy, hanem előjel: ha 0, akkor pozitív szám, ha 1, akkor negatív szám. Így azonban csak 7 bitünk marad a számról, azaz 0-tól 127-ig tudunk számot tárolni az előjelen kívüli részen. Ha 0...255 közötti számnak tekintjük a bájtot, akkor **előjel nél-**

küli egész számról beszélünk, ha pedig -127 és +127 közötti számnak, akkor **előjeles** egész számról. A valóságban általában ettől eltér az előjeles számok tárolása, -128 és +127 közötti értékek vannak az előjeles bájtban, mert nem szükséges ± 0 , és a negatív számok nem csak az előjelben különböznek. Az egészek, és a többi típus számábrázolási kérdéseivel később részletesen foglalkozunk. Most elég annyit tudnunk, hogy a számítógép mindent bájtokban tárol, legyen az egész szám, valós szám vagy szöveg.

A **processzor** ► bekapcsolás után beolvassa a memória elején lévő utasítást. Ez általában egy ugró utasítás, amely azt mondja meg, hogy hol találja a következő utasítást a memóriában, amit végre kell hajtania.

A számítógép elemi utasításai nagyon egyszerűek azért, hogy az őket megvalósító áramkörök minél egyszerűbbek, és ezáltal minél gyorsabbak lehessenek. Ha bonyolultabb utasítást akarunk adni a számítógépnek, azt elemi utasításokból építjük fel. **Elemi utasítás** az az utasítás, amit a számítógép processzora közvetlenül végrehajt, tehát már nem bontjuk kisebb utasításokra.

Kétféle elemi utasítás létezik: ugró utasítás és nem ugró utasítás. A nem ugró utasításokat a processzor sorban egymás után, azaz **szekvenciálisan** hajtja végre. A nem ugró utasítások pl. az elemi matematikai műveletek: összeadás, kivonás, szorzás, osztás, maradékképzés, negálás (=ellentétes előjelűre alakítás). Vanak összehasonlító utasítások (<, >, egyenlő, nem egyenlő, stb.), logikai utasítások (ÉS, VAGY, NEM), adatmozgató utasítások, bitműveletek, stb.

A processzor az adatokat beolvassa a memóriából, végrehajtja rajtuk az utasítás(ok) által előírt műveletet, majd az eredményt visszairja a memóriába.

Mivel a memóriában csak egész számokat tudunk tárolni, az elemi utasításokat is számként tároljuk: minden utasításhoz tartozik egy szám, ez az utasítás **gépi kódja**. Egy utasítás általában több bájt hosszúságú.

A processzor tud adatokat fogadni a perifériákról, és tud adatot küldeni a perifériáknak.

A **perifériák** ► közül számunkra mindössze három lesz lényeges: a billentyűzet, a képernyő és a háttértár. A képernyőre nem fogunk rajzolni, csak szöveget írunk ki.

1.2 A programozás alapjai

Az 1.1 pontban bepillantottunk a számítógépek felépítésének és működésének alapjaiba, alacsony szintű dolgokkal foglalkoztunk. Az 1. fejezet hátralévő részében ellenkező irányból közelítünk, és sok fontos alapfogalmat fogunk áttekinteni.

Definíciók ► Az alábbi definíciókat megjegyezni nem kell. Arra való, hogy ha bizonytalanok vagyunk valamiben, visszalapozunk.

Programozás: számítógép-algoritmusok és adatszerkezetek megtervezése és megvalósításuk valamely programozási nyelven.

Algoritmus: Valamely probléma megoldására alkalmas véges számú cselekvéssor, amelyet véges számú alkalommal mechanikusan megismételve a probléma megoldását kapjuk.

Adatszerkezet: az adatelemek egy olyan véges halmaza, amelyben az adatelemek között szerkezeti összefüggések vannak. Az adatelem az a legkisebb adategység, amelyre hivatkozni lehet.

Programozási nyelv: a számítástechnikában használt olyan, az ember által olvasható és értelmezhető utasítások sorozata, amivel közvetlenül, vagy közvetve (például: gépi kódra fordítás

után) közölhetjük a számítógéppel egy adott feladat elvégzésének módját.

Magyarázatok ► Az algoritmus definíciójában kétszer is szerepel a *véges* szó. A cselekvések számának végesnek kell lennie, mert végtelen utasítást nem tudunk leírni. Véges számú ismétlés: ez általában egyet jelent, de ha többször ismétlünk, akkor is véges sokszor, különben időben végtelen ideig tartana a művelet. Mechanikus: azt jelenti, hogy gondolkodás nélkül, vagyis egy gép is végre tudja hajtani.

Adatszerkezet esetében szerkezeti összefüggés pl. hogy egy szövegben a betűket sorban, egymás mellett tároljuk.

A programozási nyelv az ember számára áttekinthető formában történő programírást tesz lehetővé, ebből a fordítóprogram készíti a gép számára érthető gépi kódot.

Mint látjuk, az algoritmus egy általános fogalom, nem kötődik a számítógépekhez. Például, ha veszünk egy lapra szerelt könyvespolcot, általában van a dobozban egy papírlap, rajta ábrákkal, melyek az összeszerelés folyamatát, azaz az összeszerelés algoritmusát mutatják.

A számítógépprogramok **adatok** (információkat) kezelnek, dolgoznak fel. A legegyszerűbb (valamire használható) program is kezel adatokat. Ha ki akarjuk írni a számokat 1-től 10-ig, akkor tudnunk kell, éppen mennyinél tartunk, ez adat. Ha csak ki akarunk írni egy szöveget, már ott is adatot kezelünk, mert a szöveg maga is adat. Általában a programok komoly mennyiségű adatot kezelnek. Akkor lesz hatékony a program működése, ha az adatokat olyan módon tároljuk, ahogy az a legkedvezőbb, azaz megfelelő adatszerkezet(ke)t választunk.

Program = algoritmus + adatszerkezet.

Kódolás: az algoritmus és az adatszerkezet megvalósítása valamely programnyelven.

A programozás alapjainak elsajátítása ► a következő dolgokat jelenti:

- Megismertünk sok fontos alapvető algoritmust, például ilyen a prímkeresés, kiválasztás, keresések, rendezések, adatszerkezet-kezelési algoritmusok, szövegfeldolgozás, rekurzív algoritmusok, véges automata.
- Megismertünk néhány adatszerkezetet, például tömb, struktúra, fájl, láncolt listák, fák.
- Megtanulunk kódolni szabványos C nyelven.
- Megismerjük a számítógépek működésének alapjait.

A C nyelv ► sok korszerű programnyelv alapja. Sok más nyelvhez képest kevés nyelvi elemből épül fel, ezért viszonylag könnyű megtanulni. Hatékony strukturált programok írását teszi lehetővé. Tökéletesen alkalmas arra, hogy segítségével megtanuljuk az algoritmusok és adatszerkezetek gyakorlati megvalósítását.

A C nyelvet az 1970-es években fejlesztették ki, amikor a képernyő nem volt a számítógépek nélkülözhetetlen része. A szabványos C nyelv függvénykönyvtára csak olyan megjelenítő függvényeket tartalmaz, amelyek szöveges nyomtatón is működnek. C-ben nem tudunk rajzolni, de még a képernyőn pozicionálni sem, csak külső, nem szabványos függvénykönyvtárak segítségével, melyek nem képezik a tananyag részét. Felmerülhet a kérdés, hogy miért nem egy korszerű, ablakozó grafikát támogató programnyelvet tanulunk? Azért, mert ez csak elterelné a figyelmünket a lényegről: az algoritmusokról és az adatszerkezetekről. Ne feledjük:

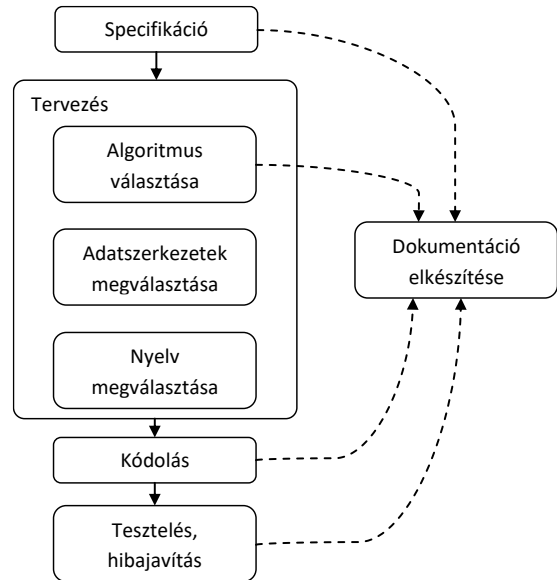
programozás != kódolás

Azaz a programozás nem egyenlő a kódolással. (A C nyelvben a nem egyenlőt !=ként írjuk, mert nincs áthúzott egyenlőségjel a billentyűzeten.) Aki tud kódolni C-ben vagy más nyelven, még nem tud programozni. Az fog tudni jó programokat készíteni, aki biztos alaptudással rendelkezik.

Ahogy említettük, a C nyelv nagy előnye, hogy sok programnyelv alapja (pl. Java, C#, de pl. a php is sokat átvett belőle), tehát az itt megszerzett tudást jól lehet használni a későbbiekben.

Eredetileg a UNIX operációs rendszer elkészítéséhez fejlesztették ki, a fő cél a hatékonyság, a minél tömörebben megírható programkód volt. Mivel nem oktatási célra fejlesztették ki, nem volt cél az, hogy a nyelv egyes részeit egymásra építve lehessen elsajátítani. Többször lesz olyan az anyagban, hogy először valamit nem magyarázunk meg, egyszerűen csak elfogadjuk, hogy ez így van, és később látjuk majd, hogy miért.

A programozás menete ►



Specifikáció: a program feladatának, bemeneti és kimeneti adatainak megadása. Ebben a könyvben sok ilyen találunk: a megoldandó feladatok szövege specifikáció. (Pl. Írjon C programot, amely...)

Tervezés: a feladat, azaz a specifikáció ismeretében választjuk ki a programnyelvet (senki sem szerkesztene weblapot C-ben...), az algoritmusokat és az adatszerkezeteket.

Kódolás: A kiválasztott nyelven megvalósítjuk a megtervezett algoritmusokat és adatszerkezeteket. Ez részben egyszerre történik a tervezéssel, azaz bizonyos algoritmikus ill. adatszerkezeti kérdésekben a kódolás közben döntünk.

Tesztelés, hibajavítás: Senki sem ír hibátlan kódot, azaz mindig van mit javítani. A hibáknak két csoportja van: szintaktikai (formai) és szemantikai (tartalmi) hibák.

Szintaktikai hiba a formai szabályok megsértése. Valamit rosszul gépelünk, lemarad egy zárójel, stb. Bár a kezdő programozónak sok bosszúságot okoznak a szintaktikai hibák, ezekkel könnyű elbánni, mert a fordítóprogram jelzi, hol találhatóak.

Szemantikai hiba esetén a fordítóprogram nem talál hibát, mert a program formailag jó, csak nem azt csinálja, amit szeretnénk, hanem azt, amire utasítottuk. Szemantikai hibára jó példa a közismert vicc:

Censored

Itt tehát a páciens szintaktikailag helyes algoritmust adott a doktornak, aki pontosan végrehajtotta, amire utasítást kapott, csak a páciens nem ezt szerette volna.

Egy program esetében nagyon fontos, hogy kiszűrjük a szemantikai hibákat, mielőtt azt valaki élesben használná. Ha nem teszteljük megfelelően, nem is fogunk tudni arról, hogy hibás.

A **dokumentálás** célja, hogy segítsük a program megértését. Szólhat a felhasználónak (használati utasítás), és szólhat a fejlesztőnek. A fejlesztői dokumentáció tartalmazza az adatszerkezetek és algoritmusok, fájlformátumok leírását, valamint a tesztelésnél használt adatokat. Legfontosabb eleme a program megjegyzésekkel ellátott forráskódja.

2. Az algoritmusok megadása

Algoritmusok megadására a programtervezés illetve dokumentálás során van szükség. Alapvetően két módon adhatunk meg algoritmusokat: szövegesen és grafikusán. Két-két megadási módról ejtünk pár szót.

- Szöveges megadás: pszeudokód, programkód.
- Grafikus megadás: folyamatábra, struktogram.

A **programkód** és a **pszeudokód** hasonlít egymásra, erre hamarosan több példát is látunk. Fő különbség, hogy a programkód megértéséhez ismerni kell az adott nyelv szabályait, míg a pszeudokód igyekszik minden programozó számára érthető lenni. A pszeudokód vonatkozásában nincsenek kötelező szabványok.

A **folyamatábra** és a **struktogram** viszonya hasonló a pszeudokód és a programkód viszonyához olyan értelemben, hogy a struktogram értelmezése és elkészítése igényel némi programozói tapasztalatot, míg a folyamatábra mindenki számára érthető. A folyamatábra hátránya, hogy nagyon könnyű vele olyan algoritmust alkotni, ami nem alakítható közvetlenül strukturált programkóddá, míg a struktogrammal leírt algoritmusok mindig strukturáltak.

Teafőző robot

Nézzünk meg egy példát egy feladat algoritmizálására! Van egy robotunk, akit szeretnénk arra utasítani, hogy főzzön teát. Hogyan utasítjuk a feladatra?

1. változat:

Főzz teát!

Ha van olyan utasítása, hogy „Főzz teát!”, akkor semmi egyéb teendőnk nincs, mint kiadni ezt az utasítást. Ha nincs ilyen utasítás, akkor meg kell magyaráznunk neki, mit is várunk el tőle.

2. változat:

„Főzz teát!” parancs:

Tölts vizet a kannába!
Kapcsold be a tűzhelyet!
Tedd rá a kannát!
Amíg nem forr
 Várj egy percet!
Dobd bele a teatojást!
Vedd le a tűzről!
Ha kell cukor
 Tedd bele!
Egyébként
 Ha kell méz
 Tedd bele!
Töltsd csészébe!

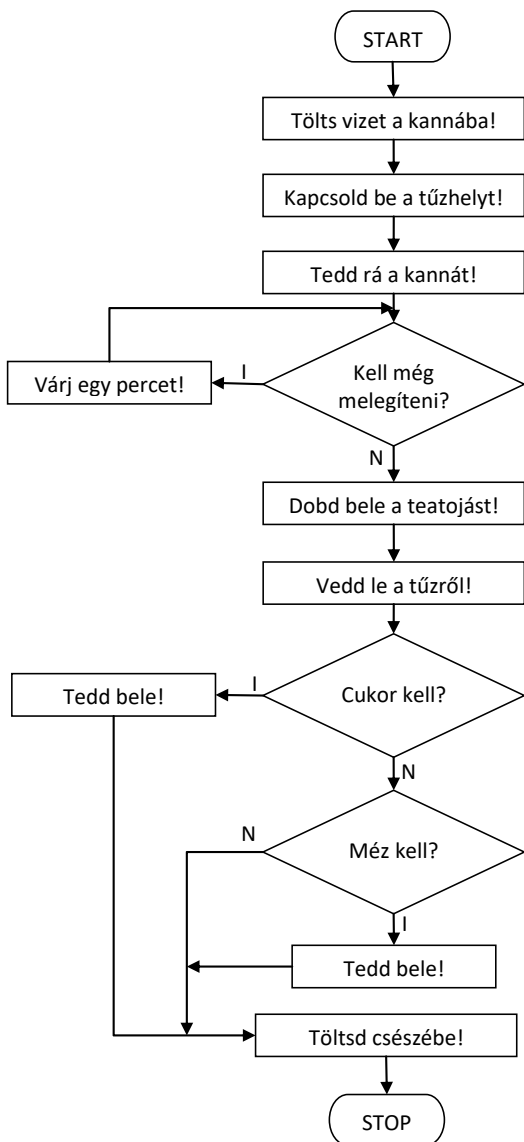
VÉGE.

Ez a teafőzés algoritmusának pszeudokódja. Amit beljebb kezdünk, az a kívül kezdődő részhez tartozik (azaz egy **blokkban** vannak). Ebben a pszeudokódban megjelenik a **strukturált programozás** mindhárom alkotóeleme:

- Ahol egy oszlopban kezdődnek az egymást követő utasítások, az a **szekvencia**.
- Az „amíg” kezdetű utasításblokk egy **ciklus**, más néven **iteráció**. A **ciklus magjában** lévő utasítás vagy utasítások addig ismétlődnek, amíg az „amíg” után álló feltétel igaz
- Feltételes **elágazás** a „ha” kezdetű utasításblokk. Ha a „ha” utáni állítás igaz, akkor végrehajtódik a feltétel magjában lévő utasítás vagy utasítások. A feltételes elágazásnak lehet (de nem kötelező) egy

„egyébként” ága is, ami akkor hajtódik végre, ha a „ha” utáni állítás hamis.

A fenti algoritmust folyamatábrával is megadhatjuk:



A folyamatábrán jól látszik a ciklus és az elágazás közötti különbség: ciklus esetén visszaugrunk, elágazásnál pedig előre haladunk.

Könnyen tudunk a folyamatábrába olyan nyilakat, azaz ugrásokat rajzolni, amelyeket a pszeudokódban, a tanult strukturált elemekkel nem tudnánk megvalósítani. Próbáljuk ki!

Megnéztük a formát, beszéljünk a tartalomról! Ha végignézi az algoritmust, valószínűleg felvetődik önben, hogy nem ön nem így főzne teát. Miért nem filtert használunk? Miért a kannába tesszük a cukrot vagy a mézet és nem a csészébe? Stb. A specifikáció nem rögzítette ezeket a részleteket. Egészen pontosan semmit sem rögzített, csak azt, hogy a robot főzzön teát. Ez alapján akár minden ízesítés elhagyható lett volna, vagy továbbákkal lehetne kiegészíteni. Az algoritmus készítője hozta meg a döntéseket úgy, ahogy neki tetszett.

Általában is elmondható, hogy a programozó dönt rengeteg algoritmikus kérdésben, a specifikáció általában nem ad választ minden kérdésre. Szükség esetén konzultálhatunk is a megbízóval, hogy mik az elvárásai.

További probléma, hogy miért ezekre a részekre bontottuk? Ez a felbontás akkor jó, ha a robot ismeri az összes benne szereplő utasítást. Ha nem ismeri, tovább kell bontanunk az algoritmust egészen addig, amíg olyan utasításokhoz nem érünk, amit a robot is ismer. A robot által ismert utasításokat **elemi utasításoknak** nevezzük, ezekből lehet bonyolultabb programokat készíteni.

Tegyük fel, hogy a robotunk nem ismeri a 2. változat utasításainak egy részét! Ha ezeket felbontjuk, akkor az algoritmus áttekinthetetlenül válik, túl hosszú lesz. Mégis muszáj felbontani az utasításokat, különben a robot nem fogja érteni, mit akarunk tőle. Van erre is megoldás: a 2. változatot így hagyjuk, és a benne található **összetett utasítások** szétbontását külön ábrázoljuk. Például a „Tölts vizet a kannába!” így nézhet ki:

„Tölts vizet a kannába!” parancs:
 Menj a csaphoz!
 Tedd alá a kannát!
 Nyisd meg a csapot!
 Amíg nincs tele
 Várj egy másodpercet!
 Zárd el a csapot!
 VÉGE.

Ennek a különválasztásnak más előnye is van: ha többször is szükség van ugyanarra a **funkcióra**, akkor mindig csak a nevét kell leírunk, a robot már tudni fogja, hogy mit kell tennie. A valódi strukturált programok nagyon fontos tulajdonsága, hogy így különválaszthatók a funkciók. Az angol **function** kifejezés funkcióként is fordítható, azonban a magyar nyelvben a **függvény** változata terjedt el. Használatos még az **eljárás**¹, vagy **szubrutin** elnevezés is. A C nyelv a függvény kifejezést használja, és látni fogjuk, hogy a matematikai függvények is valóban megadhatók ilyen formában.

Nem biztos, hogy azonnal eszünkbe jut, de a fenti algoritmust elemezve gondot jelenthet, ha pl. nincs víz, vagy, mondjuk, nem lehet megnyitni a csapot. Egy program írása során fel kell készülni a váratlan helyzetekre, és a problémákat kezelni kell. Ezt **hibakezelésnek**, vagy **kivételkezelésnek** nevezzük. A fenti esetben pl. a következőképpen járhatunk el:

„Tölts vizet a kannába!” parancs:
 Menj a csaphoz!
 Tedd alá a kannát!
 Ha megnyitható a csap
 Nyisd meg a csapot!
 Egyébként
 Mondd el a főnöknek a problémát!
 Fejezd be a programot!
 Amíg nincs tele
 Ha van víz
 Várj egy másodpercet!
 Egyébként
 Mondd el a főnöknek a problémát!
 Fejezd be a programot!
 Zárd el a csapot!
 VÉGE.

A folyamatábra készítéséhez szükséges ismeretekkel, valamint a struktogrammal a 20. fejezetben foglalkozunk.

¹ Az eljárás (procedure) kifejezést olyan esetben használják, ha nincs visszatérési érték. Ez a C-ben a void típusú függvény.

Ugyanez C nyelven:

```
#include <stdio.h>
int main(void) {
    int szam;
    szam = 1;
    while( szam <= 10 ){
        printf("%d ",szam);
        szam = szam + 1;
    }
    return 0;
}
```

A pszeudokódban és a két programban azonos sorszám jelzi az azonos jelentésű sorokat. Programok között több formai különbséget látunk, de vannak hasonlóságok is.

- Mindkét programban **definiálni** kellett a változót, ezt a részt (a)-val jelöltük. A Pascal és a C is erősen **típusos nyelv**, azaz a változókról meg kell mondani, hogy milyen fajta adatok tárolására alkalmasak. Itt integer illetve int jelöli azt, hogy egész számokat tárolhat. Például PHP-ben vagy Basic-ben nem kell definiálni a változókat, azokban bármilyen adat lehet. Ez egyszerűsíti a programírást, viszont növeli a hibalehetőséget.
- A pszeudokódban azzal jelöltük, hogy mely utasítások vannak egy másik utasítás blokkjában, hogy beljebb kezdjük (tabuláltuk) ezeket a sorokat. A Pascal és a C nyelv viszont nem írja elő, hogy beljebb kezdjünk bármit, vagy akár külön sorba kerüljenek a dolgok, ezért más módon kapcsolja össze illetve választja szét az utasításokat. Az **utasítások elválasztására** mindkét nyelvben a **pontosvessző (;)** szolgál, míg az **egy blokkba tartozó utasításokat** Pascalban a begin-end pár határolja, C-ben pedig a {} kapcsolos zárójel pár. C-ben **nincs pontosvessző** a # kezdetű sorok végén, valamint a blokk végén sem.
- Értékkadásra Pascalban a :=, míg C-ben az = **operátor** szolgál (**operátor** jelentése: **műveleti jel**). **FIGYELEM! Az értékkadás mindig jobbról balra történik!** Azaz nem jó az $i+1=i$ kifejezés! (Ellenkező esetben a fordító nem tudná, hogy egy $a=b$; kifejezésben a vagy b kap-e új értéket.)
- A kiírás szintaktikája egész más a két nyelvben. Pascalban a write függvény vesszővel elválasztva kapja a kiírandó adatokat. C-ben a printf függvény egy **formátum-sztringnek** nevezett, "" közé helyezett szöveggel kezdődik, melyben %d jelzi, hogy itt egy egész számot akarunk kiírni 10-es számrendszerben (azaz decimálisan), majd szököz, hogy szöközt akarunk kiírni. A C formátum-sztringben két speciális jelentésű karakter van, a % és a \ (fordított per jel = **back slash**). Az ezek után írt betű vagy kifejezés speciális jelentéssel bír, erről a 12. fejezetben lesz szó. Minden egyéb, amit az idézőjelek közé írunk, változatlan formában jelenik meg. (A Windows a magyar ékezetes betűket rosszul kezeli, ezek helyén egyéb karakterek jelennek meg a képernyőn, ezért ha ki akarunk írni valamit, ékezetek nélkül tegyük!)
- A C program elején és végén lévő részre az 5. fejezetben adunk magyarázatot, ebben a fejezetben minden programunk így fog kinézni:

```
#include <stdio.h>
int main() {

    Az algoritmusnak megfelelő programkód

    return 0;
}
```

A továbbiakban csak C nyelven adjuk meg a kódot. A fenti példán azt akartuk szemléltetni, hogy mennyire hasonló egymáshoz az algoritmus és a programkód, bármilyen programnyelven is valósítjuk meg. Ez azt jelenti, hogy aki tud programozni, annak egy új nyelv elsajátítása sokkal kisebb munkájába kerül, mint alapoktól megtanulni az elsőt.

2. feladat

Írjuk ki a páratlan számokat a képernyőre 1-től 21-ig!

A feladat megoldásához kiindulási alapként felhasználhatjuk az 1. feladat algoritmusát. Itt is folyamatosan növekednek a kiírt számok, viszont nem egyesével. Két algoritmust is mutatunk:

Páratlanok program 1:

```
Jegyzd meg, hogy az aktuális számunk 1!
Amíg a szám kisebb vagy egyenlő, mint 21
Ha a szám nem osztható kettővel
    Írd ki a számot!
    Írj ki egy szöközt!
    Növeld meg a számot eggyel!
```

VÉGE.

Páratlanok program 2:

```
Jegyzd meg, hogy az aktuális számunk 1!
Amíg a szám kisebb vagy egyenlő, mint 21
    Írd ki a számot!
    Írj ki egy szöközt!
    Növeld meg a számot kettővel!
```

VÉGE.

Melyik megoldás helyes?

Mindkettő helyes, mindkettő jó.

Egy problémát többféle, helyes módon is meg lehet oldani. Azt a kérdést már fel lehet tenni, hogy melyik megoldás a jobb.

Az első megoldásnál végigmegyünk minden számon, és azt vizsgáljuk, hogy a szám megfelel-e a követelménynek, azaz páratlan-e, és ha igen, kiírjuk. Az a megoldási módot, amikor minden lehetséges kombinációt kipróbálunk, **brute-force** (nyers erő) módszernek nevezzük. Gyakran használnak brute-force módszert például jelszók feltörésére: minden lehetséges jelszót végigpróbálnak, míg el nem jutnak az igazihoz. Nem hatékony, de előbb-utóbb eredményt hoz.

A második esetben viszont kihasználjuk, hogy ismerjük a páratlan számok közötti matematikai összefüggést (azaz, hogy kettővel nagyobb a következő az előzőnél), és ezáltal a ciklus lépésszámát a felére csökkenthetjük, és még az oszthatóságot sem kell vizsgálni.

Lássuk a két megoldást C nyelven:

1.

```
#include <stdio.h>
int main(void) {
    int i = 1;
    while( i <= 21 ){
        if( i % 2 == 1){
            printf("%d ",i);
        }
        i = i + 1;
    }
    return 0;
}
```

2.

```
#include <stdio.h>
int main(void) {
    int i = 1;
    while( i <= 21 ) {
        printf("%d ", i);
        i = i + 2;
    }
    return 0;
}
```

Az első program feltételes elágazásába ez a feltétel került:

```
i % 2 == 1
```

Mit jelent ez?

A dupla egyenlőségjel, azaz **==** operátor két érték egyenlőségét vizsgálja. Matematikában csak egyféle egyenlőségjelet szoktunk írni, azonban két jelentéssel. C-ben megkülönböztetjük a jelentést: $x=y$ és $x==y$ egészen mást jelent. $x=y$ azt jelenti, hogy x -nek értékül adjuk y -t, $x==y$ pedig azt vizsgálja, hogy vajon egyenlő-e x és y ?

A **% operátor** a C nyelvben a maradékképzés jele. $x \% y$ esetben x -et y -nal elosztva a maradékot kapjuk. Pl. $10\%5==0$, $11\%5==1$, $12\%5==2$, $13\%5==3$, $14\%5==4$, $15\%5==0$, stb. (Tíz osztható öttel, azaz a maradék, nulla; tizenegy nem osztható öttel, mert a maradék egy, stb.) A **% operátor csak nemnegatív egészekben** értelmezett, tehát negatív egészekben vagy valóságon nem definiált eredményt ad.

Az **==** ellentéte a **!=** nem egyenlő operátor.

Az $i \% 2 == 1$ művelet helyett írhatjuk, hogy $i \% 2 != 0$ is. Miért?

Most összevontuk az első példa (a) és (1) elemeit: egyszerre definiáljuk a változót, és adunk neki **kezdőértéket**.

Mi történne, ha elfeledkeznénk a kezdőértékről?

```
int i;
while( i <= 21 ){ // ajjjaj!
    printf("%d ", i);
    i = i + 2;
}
```

Ez a programrészlet hibásan működik, és működése előre kiszámíthatatlan. Amíg egy változó nem kap értéket, **inicializálatlan**. A C-ben egy inicializálatlan változó tartalma meghatározatlan, bármi lehet benne. Ezt a bármit **memóriaszemétnak** hívjuk, mert minden változó van valahol a memóriában, és amikor a rendszer kiosztja a memóriát a változóknak, nem állít be semmilyen értéket. A változók értéke az lesz, ami az adott helyen korábban volt.

3. feladat

Az előző feladatban rövidebb és egyszerűbb, ráadásul hatékonyabb is a második megoldás, és inkább ez jut az ember eszébe, semmint a másik. Nézzünk egy olyan esetet, amikor a brute-force algoritmus a kézenfekvő, és gondolkodni kell a hatékonyabb megoldáshoz!

Írjuk ki két pozitív egész szám legnagyobb közös osztóját!

A feladat megoldásához ismerjük az összes algoritmikus eszközt és a C nyelv összes olyan elemét, ami szükséges az algoritmus leködöléséhez, mégsem tűnik egyszerűnek a feladat.

Ha azonban végiggondoljuk az előző feladat brute-force megoldását, már nem is olyan nehéz a feladat:

- Válasszuk ki a két szám közül a kisebbet, mert ez lehet a legnagyobb olyan érték, ami közös osztója lehet a két számnak.
- Nézzük meg, hogy ez osztója-e mindkét számnak. Ha igen, megvagyunk. Ha nem, akkor csökkentünk eggyel a számot, és ezt vizsgáljuk meg, hogy osztója-e mindkettőnek.
- Ismételjük a csökkentést és a vizsgálatot, amíg közös osztót nem találunk. Biztosan lesz ilyen, mert 1 minden számnak osztója.

Írjuk le az algoritmus pszeudokódját!

LNKO 1 program:

Kérj be két pozitív egész számot!

Ha az első kisebb, mint a második

Jegyezd meg, hogy az osztó az első!

Egyébként

Jegyezd meg, hogy az osztó a második!

Amíg osztó nem osztója mindkét számnak

Csökkentsd osztó értékét

Írd ki osztót!

VÉGE.

C nyelven:

```
#include <stdio.h>
int main(void) {
    int szam1, szam2, oszto;
    printf("Adj meg 2 poz. egészt: ");
    scanf("%d%d", &szam1, &szam2);
    if(szam1 < szam2) {
        oszto = szam1;
    }
    else {
        oszto = szam2;
    }
    while(!(szam1%oszto==0
        && szam2%oszto==0)){
        oszto = oszto - 1;
    }
    printf("LNKO = %d", oszto);
    return 0;
}
```

A számok beolvasásához a scanf függvényt használjuk. A függvény használata hasonlít a printf-ére: egy formátumsztring határozza meg, hogy hány és milyen típusú értéket kell megadnia a **felhasználónak** (vagyis annak, aki a kész programot használja), ezt követően, vesszővel elválasztva következnek a változók. A változók neve elé egy **&** jelet is kell írni! A felhasználónak szóközt vagy ENTER-t kell írnia a számok közé.

Jegyezzük meg, hogy a scanf nem ír ki semmit. Ha a %d-ken kívül mást is írunk ide, akkor a scanf azt várja, hogy a felhasználó is beírja ezt a szöveget. Ha a felhasználó nem írja be, akkor hibásan működik. Pl. ha scanf("Szia%d",&szam1);-t írunk, akkor a felhasználónak így kellene beírnia a számot: *Szia120*. Ha a felhasználó csak a 120-at írja be, a scanf ezt nem fogadná el. A scanf-ről bővebben a 12. fejezetben lesz szó.

A while ciklus feltétele ezúttal jóval bonyolultabb, mint korábban. A **!(feltétel)** a feltétel **negálását** jelenti, azaz IGAZ-ból HAMIS-at, HAMIS-ból IGAZ-at csinál. Jelen esetben a zárójelben akkor van IGAZ, ha mindkét szám osztható az *osztóval*, vagyis *osztó* közös osztójuk, minden más esetben HAMIS. A ciklusnak viszont addig kell ismétlődnie, amíg *osztó* nem közös osztója a két számnak. A **!**-lel jelölt műveletet **logikai NEM**-nek is mondjuk.

A zárójelben álló két feltétel közé az **&&** operátor került, ennek neve **logikai ÉS** kapcsolat. Az ÉS művelet eredménye akkor IGAZ, ha mindkét feltétel IGAZ, bármelyik is HAMIS, az eredmény HAMIS.

A két megismert logikai művelet igazságtáblája a következő:

A	!A
HAMIS	IGAZ
IGAZ	HAMIS

A	B	A&&B
HAMIS	HAMIS	HAMIS
IGAZ	HAMIS	HAMIS
HAMIS	IGAZ	HAMIS
IGAZ	IGAZ	IGAZ

A `szam1%osztó==0 && szam2%osztó==0` kifejezést szóban így mondhatjuk: „*szam1* osztható *osztó*val, és *szam2* is osztható *osztó*val”. A hétköznapi beszédben ezt mondhatnánk: „*szam1* és *szam2* is osztható *osztó*val”, azonban ha programot írunk, nem írhatjuk így: „`szam1 && szam2%osztó==0`”!



A C nyelvben összesen három logikai művelet van: a NEM, valamint ÉS mellett a harmadik a **logikai VAGY** kapcsolat. A logikai VAGY kapcsolatot C-ben **||** operátor jelöli. Ez két függőleges vonal, mely magyar billentyűzetben a W+Alt Gr kombinációval érhető el (gyakran így ábrázolják: |). Ennek igazságtáblája a következő:

A	B	A B
HAMIS	HAMIS	HAMIS
IGAZ	HAMIS	IGAZ
HAMIS	IGAZ	IGAZ
IGAZ	IGAZ	IGAZ

A VAGY kapcsolat akkor ad igazat, ha bármelyik, de akár mindkét állítás igaz. Csak akkor hamis, ha mindkét állítás hamis. Figyelem, a hétköznapi életben másképp használjuk a „vagy” szót, pl. almát vagy körtét fogok enni: ez azt jelenti, hogy vagy egyiket, vagy másikat, de nem mindkettőt. A logikai VAGY kapcsolat viszont mindkettőt is megengedi. A számítástechnika azt a vagy kapcsolatot, amikor a két állításból pontosan egy IGAZ, akkor IGAZ az eredmény, KIZÁRÓ VAGY kapcsolatnak nevezi. A C nyelvben nincs ilyen **logikai művelet**, de találkozni fogunk vele a **bitműveleteknél**.



A VAGY művelet felírható ÉS és NEM műveletek segítségével, az ÉS pedig felírható VAGY és NEM műveletek segítségével. Ezek a De Morgan azonosságok:
 $A||B \equiv !(! (A) \ \&\& \ ! (B))$,
 $A\&\&B \equiv !(! (A) \ || \ ! (B))$.
 Igazolja igazságtábla felírásával!

A while ciklus feltételét így is írhattuk volna:
`while(szam1%osztó!=0 || szam2%osztó!=0)`
 Gondolja át, miért!

A C-ben a műveleti jelek és a változók közé **szabad szóközt tenni**, de ez nem kötelező. Az a cél, hogy a programkód minél jobban érthető legyen az ember számára, a fordítóprogram így is úgy is megérti. A C nyelv a **szóközt, tabulátort, újsor jelet** (ENTER), és más, kevésbé használt „nem látható” karaktereket egyformán kezeli: bármelyikből bármennyi írható oda, ahová

írható egy. Ezeket a karaktereket összefoglaló néven **whitespace** karaktereknek nevezzük. Ennek értelmében az if utasítás blokkja így is írható pl.: `{osztó=szam1;}`. Azért írtuk úgy, ahogy, hogy jobban hasonlítson a pszeudokódra.

Javíthatjuk az algoritmus hatékonyságát, ha először megnézzük, hogy a kisebbik szám lehet-e a LNKO, és ha nem, a vizsgálatot a kisebbik szám felétől folytatjuk. (A LNKO osztója a kisebbik számnak is, azaz lehet szám, szám/2, szám/3, szám/4,..., és legrosszabb esetben szám/szám=1.) Dolgozza át a fenti algoritmust ennek figyelembevételével, és készítse el a kódot C nyelven!

LNKO 1 program már tartalmazott némi megfontolást: felülről lefelé indultunk el. Lehetne ennél butább módszerrel is keresni a LNKO-t. Induljunk 1-től, és menjünk a kisebbik számig egyesével! Minden lépésben ellenőrizzük, hogy mindkét szám közös osztójáról van-e szó, és ha igen, jegyezzük meg az osztót! A kisebbik szám elérésekor az utoljára megjegyzett osztó a LNKO. Írja meg az algoritmus pszeudokódját és C nyelvű megvalósítását!

A while és if utasítások esetében, ha az utasítás magja egyetlen utasítást tartalmaz, a kapcsos zárójel elhagyható, emiatt a fenti program rövidebben is írható:

```
#include <stdio.h>
int main(void) {
    int szam1, szam2, osztó;
    printf("Adj meg 2 poz. egész: ");
    scanf("%d%d", &szam1, &szam2);
    if(szam1<szam2) osztó = szam1;
    else osztó = szam2;
    while(!(szam1%osztó==0
            && szam2%osztó==0))
        osztó = osztó + 1;
    printf("LNKO = %d", osztó);
    return 0;
}
```

Aki bizonytalan, nyugodtan írja ki a kapcsos zárójelet.

Második megoldás

Az előzőnél optimálisabb megoldást kapunk, ha a számok prímtényezősz felbontásából határozzuk meg a legnagyobb közös osztót. A prímek keresésénél továbbra is megmaradunk a brute-force módszernél.

Hatékonyabb prímkérésés valósítható meg Eratoszthenész szita módszerével, ehhez azonban tárolni kell a számokat, de legalább a megtalált prímeket, ehhez jelenleg nem rendelkezünk elegendő ismerettel.

A módszer a következő:

- Lnko 1-ről indul, az osztó 2-ről.
- Ha mindkét szám osztható osztóval, lnko-t megszorozzuk osztóval, a számokat elosztjuk osztóval, és a b) lépést ismétljük. Ha nem osztható mindkettő, megyünk tovább c)-re.
- Növeljük osztót eggyel. Ha nem értük el a kisebbik számot, b)-vel folytatjuk, egyébként d)-vel.
- Kiírjuk lnko-t.

Az imént prímtényezősz felbontásról volt szó. Hol vannak itt prímek? A két szám leosztása mindig prímeikkel történik, mert mire pl. 10-ig jutunk, addigra leosztottuk 2-vel és 5-tel őket, így már nem oszthatók 10-zel, vagy egyéb összetett számmal. Így a gyakorlatban mindig prímeikkel osztunk.

Például:

	szám1	szám2	lnko	osztó
Kezdetben	20	30	1	2
2 osztója mindkettőnek	20→10	30→15	1→2	2
2 már csak az egyiknek	10	15	2	2→3

osztója				
3 csak az egyiknek osztója	10	15	2	3→4
4 nem osztója egyiknek sem	10	15	2	4→5
5 mindkettőnek osztója	10→2	15→3	2→10	5
5>2, ami a kisebbik szám	2	3	10→	5
			kiírjuk	

A fenti algoritmus pszeudokódja a következő lesz:

LNKO 2 program:

```
Inko=1, osztó=2
Kérj be két pozitív egész számot!
Ha a második kisebb, mint az első
Cseréld fel a két számot!
Amíg az első szám nagyobb vagy egyenlő az osztóval
Mindkét számot oszd el osztóval!
Mindkét számot oszd el osztóval!
Inko-t szorozd meg osztóval!
Növeleld osztót eggyel!
Írd ki osztót!
VÉGE.
```

Az algoritmus C nyelven:

```
#include <stdio.h>
int main(void){
    int szam1, szam2, oszt=2, lnko=1;
    printf("Adj meg 2 poz. egészt: ");
    scanf("%d%d", &szam1, &szam2);
    if(szam1>szam2){ /* csere jön */
        int temp=szam1;
        szam1 = szam2;
        szam2 = temp;
    }
    while( szam1>=oszt ){
        while (szam1%oszt==0
            && szam2%oszt==0){
            szam1 = szam1 / oszt;
            szam2 = szam2 / oszt;
            lnko = lnko * oszt;
        }
        oszt = oszt + 1;
    }
    printf("LNKO = %d", lnko);
    return 0;
}
```

A C kódban /* és */ közé írhatunk megjegyzéseket. Ezekkel a fordító nem foglalkozik, nekünk viszont segít a kód megértésében. A * **operátor** a szorzás.

A fenti kódban újdonság a csere algoritmus is:

Csere algoritmus:

```
Tedd ideiglenes változóba az egyik cserélendő értéket!
A másik értéket tedd az egyik helyére!
Az ideiglenes változóba mentett értéket tedd a másikba!
VÉGE.
```

Például:

	szám1	szám2	temp
Kezdetben	30	20	?
temp=szam1	30	20	→30
szám1=szam2	→20	20	30
szám2=temp	20	→30	30

Ha ismerjük két szám legnagyobb közös osztóját, hogy hatá-

rozzuk meg a legkisebb közös többszörösüket?²

Gondolkodjunk együtt!

G3.1 Írjon algoritmust, amely 100 és 200 között kiírja az összes 3-mal osztható számot! Írja fel az ehhez tartozó C nyelvű kódot! (Nem kell teljes programot írnia!)

G3.2 Írja fel a következő műveletek igazságtábláját! Elhagyható a zárójel?

- A && B && C
- A || B || C
- A && (B || C)
- (A && B) || C

G3.3 Írja fel a következő műveletek igazságtábláját! Melyik a KIZÁRÓ VAGY művelet?

- (A || (!B)) && ((!A) || B)
- (A && (!B)) || ((!A) && B)
- !((!A) && (!B))
- !((!A) || (!B))

G3.4 Írjon algoritmust pszeudokódban, amely kiírja egy pozitív egész szám prímtényezős felbontását! Írja le az algoritmus C nyelvű megvalósítását is!

Például 30 prímtényezős felbontása:

```
30|2
15|3
5|5
1|
```

G3.5 Írja le az összeadás algoritmusának pszeudokódját! (Nem kell C kód.)

Pl.:

```
123
+348
----
471
```

G3.6 A következő pszeudokód egy szám faktoriálisát számolná ki, de van benne egy hiba. Javítsa ki a hibát! (Pl. 5 faktoriális: 1*2*3*4*5=120.)

Faktoriális program:

```
Szorzó legyen 1, Eredmény legyen 1!
Kérj be egy pozitív egész számot!
Amíg szorzó nem nagyobb a számnál
Szorozd meg Eredményt a Szorzóval!
Írd ki az Eredményt!
```

VÉGE.

G3.7 Adja meg azt az algoritmust pszeudokódban, amely megszámlolja, hogy egy pozitív egész számnak hány valódi osztója van! (Valódi osztó: 1-en és önmagán kívül az összes osztó.) Írja le az algoritmust megvalósító C nyelvű kódrészletet!

G3.8 Írjon kódrészletet C nyelven, amely kiírja, hogy egy pozitív egész szám prím-e, vagy sem! A megoldásban felhasználhatja a G3.6 feladat megoldását is!

² LKKT=(Szam1*Szam2/LNKO). Számábrázolási okokból célszerű a műveletet LKKT=(Szam1/LNKO)*Szam2 formában írni.

G3.9 Írjon algoritmust a legkisebb közös többszörös meghatározására

- brute-force módon!
- prímtényező felbontás segítségével!

Oldja meg egyedül!

Az integrált fejlesztőkörnyezet

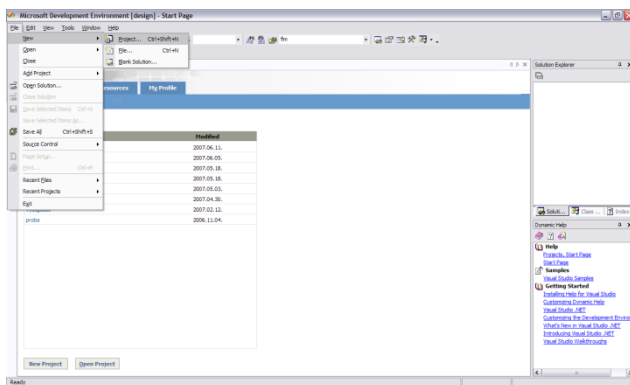
A kényelmes és gyors munka érdekében programfejlesztésre lehetőség szerint olyan szoftvereket használunk, mely egybeépítve tartalmazza a programkód szerkesztésére alkalmas szövegszerkesztőt, a programmodulokból gépi kódot létrehozó fordítót (compiler), a gépi kód modulokból futtatható programot létrehozó linkert, és a programhibák felderítését megkönnyítő debugger rendszert. Az ilyen rendszereket **integrált fejlesztőkörnyezetnek (Integrated Development Environment – IDE)** nevezzük.

A könyvben található példák kipróbálásához bármilyen C fordító használható, nem kötelező a bemutatott programokkal dolgozni.

A Microsoft Visual C++ 2003 fejlesztőkörnyezettel ismerkedünk meg röviden. A MS Visual C++ fordító Express Edition nevű, csökkentett képességű változata ingyenesen letölthető a Microsoft honlapjáról, és a vele készült programok tetszőleges célra felhasználhatók. Az Express Edition tartalmazza a számunkra szükséges funkciókat, használata nagyon hasonló a 2003-as változathoz.

Program létrehozása

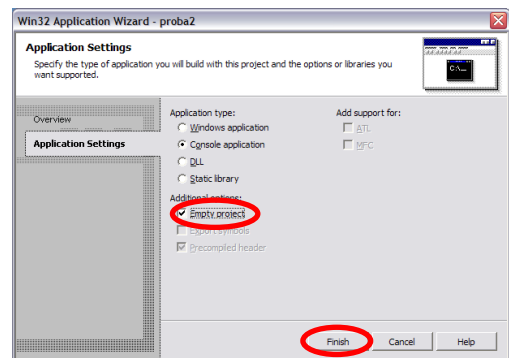
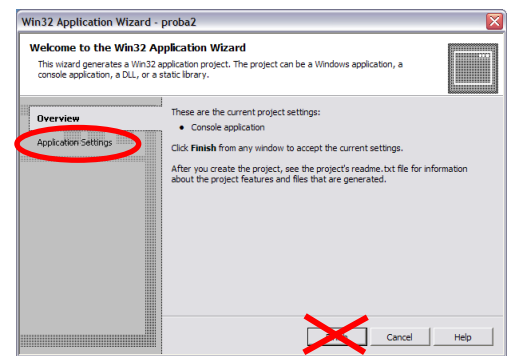
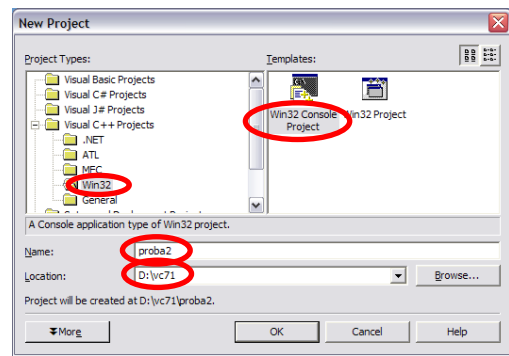
VC++ 2003-at elindítva a következő ablak fogad:



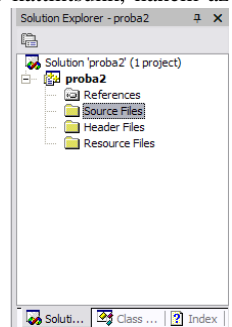
Létre kell hozni a program projektjét. Ehhez kattintsunk a File menü New/Project... parancsára!

Ne használjuk a New ikont! Ez egy txt fájlt hoz létre, amivel nem tudunk mit kezdeni, mert nem része a projektnek. Ha véletlenül mégis egy ilyen ablakba kezdtük írni a programot, ne essünk kétségbe, hanem mentjük el a fájlt (c vagy cpp kiterjesztéssel! tehát pl. valami.c), csukjuk be, majd a következőkben leírtak szerint hozzuk létre a projektet. Ha ez megvan, a jobb oldali Solution Explorerben a Source Files mappára kattintsunk jobb egérgombbal! Ezután Add/Add Existing Item → válasszuk ki az elmentett c vagy cpp fájlt!

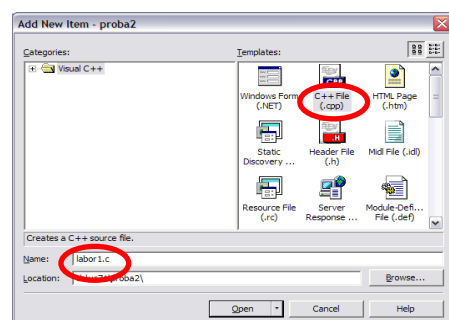
A következő dialógusablakokat kell helyesen beállítani. Figyelem, könnyű elrontani!



Figyeljünk, hogy **Win32 Console Projectet** indítsunk, és a második dialógusablakban **ne a Finishre** kattintsunk, hanem az **Application Settings** fülre! Itt pipáljuk ki az **Empty project** opciót! Ha jól csináltuk, akkor a jobb oldali ábrához hasonlóan, az IDE jobb oldalán található Solution Explorer mindegyik mappája üres lesz. Most



- jobb klikk a Solution Explorer Source Files mappáján
- Add New Item
- Válasszuk a C++ File-t, és adjunk neki nevet. Ha c kiterjesztést írunk (valami.c), akkor a fordító nem C++ fájlnak fogja kezelni.



Fordítás, futtatás, hibakeresés

Miután begépeltük a C nyelvű programot, abból futtatható programot kell készíteni (Microsoft operációs rendszerek alatt ezek .exe kiterjesztésűek). A C nyelvben a programok gyakran több forrásfájlból állnak, az is előfordulhat, hogy ezeket a modulokat más-más ember készíti. A C nyelv így támogatja a csoportmunkát. Emiatt a felépítés miatt a futtatható állomány létrehozása két lépésből áll:

1. Fordítás: ekkor minden egyes .c (vagy .cpp) fájlból egy lefordított object fájl jön létre (általában .obj kiterjesztéssel).
2. Szerkesztés (linkelés): az object fájlkat, és a rendszerfüggvényeket tartalmazó .lib fájlkból a programban használt függvényeket egybeszerkeszti, és ezzel létrehozza a futtatható programot.

Lehetőség van arra is, hogy több object fájlból mi magunk hozunk létre függvénykönyvtárakat, vagyis .lib fájlkat, ezzel a kérdéssel azonban ebben a jegyzetben nem foglalkozunk.

A fordítással összefüggő parancsok a **Build menüben**, illetve az eszköztáron is megtalálhatók (ha a Build eszköztár be van kapcsolva).

- Compile: lefordítja a .cpp fájlkat, obj. fájlkat hoz belőle létre.
- **Build Solution/Build proba2**: lefordítja azokat a forrásfájlokat, melyek módosultak a legutóbbi fordítás óta, majd a linker létrehozza az .exe-t. Egy Solutionön belül több program/library is lehet, a Build Solution az összeset lefordítja, a Build proba2 csak a kiválasztott proba2 projektet. Mi kizárólag egyprogramos Solutiont használunk, tehát a fentiek közül bármelyiket használhatjuk.
- Rebuild Solution/Rebuild proba2: a forrásfájlokat akkor is újrafordítja, ha nem módosultak a legutóbbi fordítás óta.

Amennyiben nem írjuk át a beállításokban, a fordítás a program tartalmazó mappa alkönyvtárában jön létre. Az alkönyvtár neve a fordítás módjától függ. A felső eszköztáron választhatunk **Debug** és **Release** közül, de magunk is létrehozhatunk fordítási profilt.

- **Debug**: minden optimalizáció ki van kapcsolva, a kódba kerülnek olyan részek, melyek lehetővé teszik a **hibakeresést** (töréspontok beépítése, soronkénti futtatás stb.). Emiatt ez így kapott exe nagy és lassú lesz.
- **Release**: a véglegesnek szánt változat, optimalizációval (gyorsabb programot kapunk) és a debuggolást segítő kódok nélkül: release állásban nem tudunk debuggolni, csak ha megváltoztatjuk a konfigurációt, de akkor már inkább a debug módot használjuk.

Ha bepillantunk a Debug és Release mappákba fordítás után, az .obj és az .exe fájlkon kívül számos egyebet is találunk itt. Ezekre a fájlokra a később nem lesz szükségünk, tehát nyugodtan törölhetők. Ha újrafordítjuk a programot, ismét létrejönnek.

A konfigurációkhoz tartozó beállításokat a Project menü Properties pontjában állíthatjuk be, de ugyanez a dialógusablak előhívható a Solution Explorerben a program nevének jobb egérgombot nyomva, és a Properties választva (Figyelem! Ne a Solutionön, hanem a program nevének kattintsunk, különben más Properties ablak jön elő!). Itt teljesen átszabhatjuk a beállításokat, akár olyannyira, hogy Release üzemmódban legyen olyan, mint alapértelmezés szerint Debug módban, és fordítva

Fordítási hibák

Amennyiben a programunk szintaktikai hibát tartalmazott, tehát valamit rosszul írtunk, esetleg lefelejtettünk egy zárójelet vagy pontosvesszőt, az erre vonatkozó hibaüzenetek a képernyő alján jelennek meg. A hibaüzenetre kattintva a kurzor arra a sorra ugrik, ahol a hibát elkövtük a fordító szerint, de előfordulhat, hogy a hiba az előző sorban volt, pl. ott hagytuk le a pontosvesszőt. **Ha több hibát talál a fordító, akkor mindig fölülről lefelé haladjunk ezek kijavításában**, mert gyakran előfordul, hogy a lejjebb leírt hibák nem is hibák, hanem csak egy előbb lévő hiba

következtében mondja azt a fordító. Ilyen például akkor fordul elő, ha leahagytunk egy zárójelet.

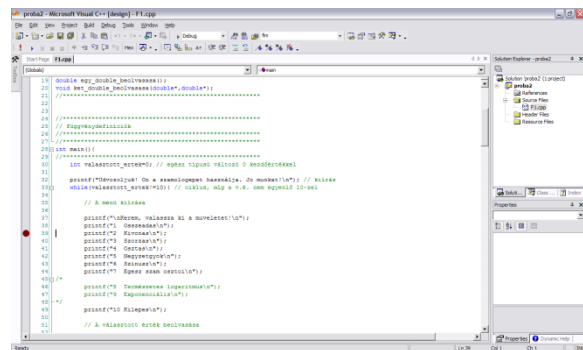
Szemantikai hibák

A következő leírásban olyan információk is találhatóak, melyekhez most még nem rendelkezünk elég ismerettel, de ha tapasztaltabbak leszünk, érdemes lesz visszalapozni.

Ha sikerült lefordítani a programot, az még nem jelenti azt, hogy helyesen is működik. Vannak olyan esetek, amikor a program szintaktikailag helyes, tehát le lehet fordítani, de a fordító talál olyan gyanús részeket, ahol hiba lehet. Ilyen esetekben figyelmeztetést (warning) ír ki. A warningokat is nézzük át, és csak akkor hagyjuk figyelmen kívül, ha biztosak vagyunk benne, hogy jó, amit írtunk.

A kezdő programozónak gyakran a szintaktikai hibák kijavítása is nehéz feladat, de ők is rá fognak jönni, hogy a szemantikai hibák (bugok) felderítése sokkal nehezebb, hiszen itt nem áll rendelkezésre a fordító segítsége, nem mondja meg, hogy itt és itt van a hiba, hanem a rendellenes működésből nekünk kell rájárnunk, hogy baj van. A fejlesztőkörnyezet azonban nem hagy ilyen esetekben sem eszközök nélkül.

Helyezzünk **töréspontot** a programba! A töréspont azt jelenti, hogy a program fut normálisan a töréspontig, és ott megáll. Ekkor meg tudjuk nézni a változók aktuális értékét, és soronként tovább tudjuk futtatni a programot (vagy normálisan is tovább futtathatjuk). Töréspontot legegyszerűbben úgy helyezhetünk a programba, ha a programkód melletti szürke (VC) sávon kattintunk bal egérgombbal. Ekkor egy piros pötty jelenik meg a sor mellett.



Start és Start Without Debugging: A Debug menüben ezzel a két paranccsal indíthatjuk a programot. Az első választva a program megáll az első töréspontnál, a második választva nem veszi figyelembe egyik töréspontot sem! Viszont ha a Start-ot választjuk, a **program ablaka becsukódik** a futás végén, nem látjuk, mit írt ki a programunk, míg Start Without Debugging esetén **nem csukódik be az ablak**.

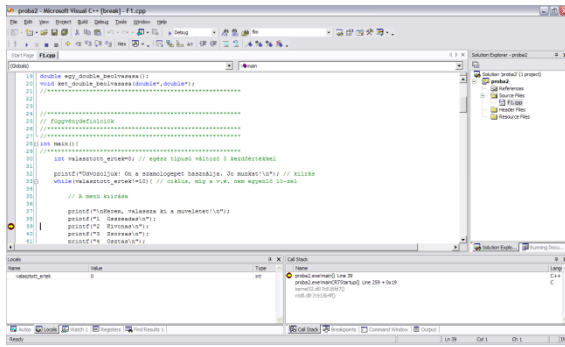
Visual C++ esetén a töréspontnál azt is megadhatjuk, hogy ne mindig álljon meg ott a program, hanem csak valahányadik odaérés alkalmával, vagy pedig valamiféle feltétel teljesülése esetén (pl. egy változó értéke valamennyi). Ezek beállíthatók a töréspontra jobb egérgombbal kattintva, a Breakpoint Properties... menüpontot választva.

A hibakeresés megkezdése után (pl. ha a program megáll egy töréspontnál) alul megjelennek a debug ablakok, ahol **megnézhetjük** és követhetjük a **változók értékét**. Ezek közül az Autos a rendszer által önkényesen kiválasztott változókat mutatja, a Locals az adott függvényben definiált összes változót, a Watch pedig az általunk megadott változókat. Ezen kívül, ha **az egeret valamelyik változó fölé** visszük, egy buborékban megjelenik a változó értéke.

A jobb alsó ablak Call Stack füle még nagyon hasznos számunkra, mert ennek segítségével tudhatjuk meg, mely függvény hívta meg az aktuális függvényt, azt melyik hívta, stb.

Nem csak változók, hanem kifejezések is beírhatók a Watch ablakba, például `t[23]`, `*yp` vagy `a+b` is.

Ha egy pointerrel adott tömb elemeit szeretnénk látni, a debugger a +ra kattintva csak a tömb első elemét írja ki. Ha pl. t a tömb neve, írjuk be a Watch ablakba, hogy t,100! Ekkor a tömb 100 elemét fogja kiírni.



Töréspont helyett a Run to Cursor (Futtatás a Kurzorig) parancsot is használhatjuk adott helyen történő megállásra.

Ha megállt a program, többféleképpen is továbbléphetjük. A soronkénti léptetésre két lehetőség is van:

- **Step Over:** végrehajt egy sornyi programot. Ha a sorban függvényhívás történik, a függvényt is végrehajtja.
- **Step Into:** elkezd végrehajtani egy sor programot. Ha a sorban függvényhívás történik, beugrik a függvénybe, és a következő Step Over vagy Step Into utasításokkal a függvényen belül lépegethetünk.
- **Step Out:** végrehajtja az adott függvényt, és visszaugrik a hívás helyére

A programok szállítása

A Debug és a Release mappát minden szívfájdalom nélkül törölhetjük. Ha kell az exe, akkor azt azért előbb másoljuk át valahova. Amikor legközelebb újrafordítjuk a programunkat, ezek automatikusan ismét létrejönnek.

A projekt főkönyvtárából a .cpp és .h fájlokra van szükségünk. Ha a többi letöröljük, akkor legközelebb ismét létre kell hoznunk a konzol alkalmazás projektet a korábban bemutatottak szerint, majd a projekthez hozzá kell adnunk a .cpp és .h fájlokat.

Ha tehát haza akarjuk küldeni napi munkánkat, akkor célszerű először letörölni a Debug és Release mappát, majd a maradékot becsomagolni (Total Commanderrel Alt+F5), és ezt csatolni a levélhez. Ha nagyon szűkös a sáv szélességünk, akkor csak a .cpp és .h fájlokat tömörítsük!

L3.1 Gépelje be (ne copy-paste!), fordítsa le és futtassa az alábbi C programot! Mi történik, ha ékezetes betűket ír? A printf kezdetű sort másolja be egymás után többször, és futtassa újra a programot! Mi történik? És ha kitörli a sztringből a \n karakterpárt?

```
#include <stdio.h>
int main(void) {
    printf("Udv, programozo!\n");
    return 0;
}
```

L3.2 Írjon C programot, amely kiírja a számokat 1-től 20-ig egyesével! (Vagy módosítja az előző programot, vagy új projektet hoz létre. Ha az előző projekthez ad egy másik C fájlt, akkor figyeljen arra, hogy egy projektben csak egy darab main függvény lehet, ezért amelyiket most nem használja, nevezze át másra!) Próbálja ki az IDE hibakereső szolgáltatásait! Helyezzen be töréspontot, és lépésenkénti futtatással figyelje a változó értékét!

L3.3 A következő program írójának billentyűzetéről hiányzott a pontosvessző. Segítsen neki, és tegye be a hiányzó ;-ket!

Használja a fordítóprogramot a hiányzó pontosvesszők megtalálásához! (Most használhat copy-paste-t.)

```
#include <stdio.h>
int main(void) {
    int szam1, szam2, oszto=2, lnko=1
    printf("Adj meg 2 poz. egész: ")
    scanf("%d%d", &szam1, &szam2)
    if(szam1>szam2) {
        int temp=szam1
        szam1 = szam2
        szam2 = temp
    }
    while( szam1>=oszto ) {
        while (szam1%oszto==0
            && szam2%oszto==0) {
            szam1 = szam1 / oszto
            szam2 = szam2 / oszto
            lnko = lnko * oszto
        }
        oszto = oszto + 1
    }
    printf("LNKO = %d", lnko)
    return 0
}
```

L3.4 A C programokban a szóközők, tabulátorok és az újsor (ENTER) karakterek azonos jelentésűek (ezeket whitespace karaktereknek nevezzük). Kivétel: a # kezdetű sorok, valamint az "..." közötti szövegek belseje: ezeket nem szabad ENTER-rel darabolni. A műveleti jelek és írásjelek előtt és mögött lehet whitespace karakter, de szorosan egymás mellé is írhatók.

Az L3.3-as feladatban található programba tegyen be illetve töröljön whitespace karaktereket úgy, hogy a program működőképes maradjon!

L3.5 A javított 1.3.3-as program működését kövesse debuggerrel!

L3.6 Írjon programot C nyelven, amely bekér a felhasználótól egy pozitív egész számot, és kiírja az összes osztóját!

4. Hogy működik a fordító?

Előfeldolgozó nincs fájl kimenete
Fordító .obj ill. .o fájlok (asm)
Linker futtatható állomány (.exe)

Az előfeldolgozó (preprocesszor) a kód takarítását (megjegyzések, felesleges szöközők törlése, stb.) és a # kezdetű sorok feldolgozását végzi, konstansokat és makrókat helyettesít.

A fordító a C nyelven megírt kódból a számítógép számára emészthető gépi kódot készít, melyet tárgykódú, ún. object fájlba ment. A gépi kód olyan elemi utasításokból álló sorozat, amely már kellően kicsi, egyszerű műveleteket tartalmaz ahhoz, hogy azt a számítógép processzora fel tudja dolgozni. Beállítható, hogy assembly nyelvű eredményt is produkáljon. Az assembly nyelv a gépi kód emberek számára jobban fogyasztható változata; az assembly kódban a processzor minden utasításához rendelnek egy jól megjegyezhető nevet (pl. mov=adatmozgatás, add=összeadás).

Egy projekt több .c fájlt is tartalmazhat, mindegyikből létrejön egy-egy object fájl. A linker ezeket egyesíti, továbbá hozzászerkeszti a szabványos függvények (pl. printf) gépi kódját. Így létrejön a futtatható állomány (Windowsban exe).

5. A C programok felépítése

Ebben a fejezetben először megismerkedünk a néhány szintaxisleíró módszerrel. Ezt követően végignézzük a legnagyobb közös osztós programunk módosított változatát, és megbeszéljük azokat a részeket, amelyeket eddig még nem.

5.1 A szintaxis leírása

Egy programozási nyelv formai szabályainak leírására többféle megadási módszer létezik. Szöveges megadásra a legelterjedtebb leírónyelv a Backus-Naur Form (BNF), illetve ennek továbbfejlesztett változata az Extended BNF(EBNF), grafikus megadásra pedig a szintaxis gráf.

BNF (Backus-Naur Form):

Elemi:

- $\langle \text{szimbólum} \rangle ::= \text{kifejezés}$
 - $\langle \text{szimbólum} \rangle ::= -\text{től balra nemterminális (felbontható)}$
- Kifejezés:
 - Egy vagy több (terminális vagy nemterminális) szimbólum
 - Egymásutániság: több, szóközzel elválasztott szimbólum
 - Választás (valamelyik a felsorolásból áll az adott helyen): |

Pl.:

$\langle \text{szám} \rangle ::= \langle \text{számjegy} \rangle | \langle \text{számjegy} \rangle \langle \text{szám} \rangle$

$\langle \text{számjegy} \rangle ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"$

EBNF (Extended BNF)

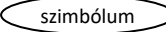
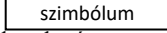
C-szerűbb leírás, több nyelvi elem.

szám = számjegy , {számjegy};

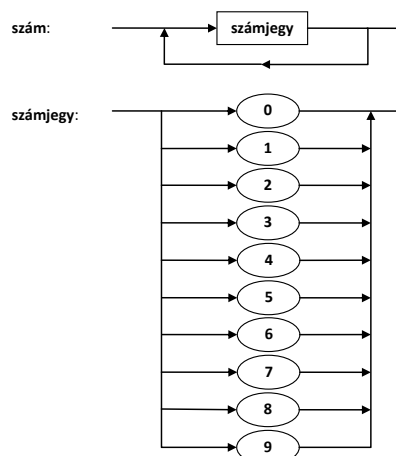
számjegy = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

Szintaxis diagram

A BNF/EBNF látványosabb verziója, irányított gráffal írjuk le a nyelvet.

- kezdet, vég: ○
- terminális szimbólum: 
- nemterminális szimbólum: 

A nem terminális szimbólum tovább bontható, a terminális nem.



5.2 Egy egyszerű C program elemei

A második fejezetben beszéltünk arról, hogy a programokat célszerű kisebb, áttekinthető részekre darabolni, és egyes részeket külön megvalósítani. Ez már az egyszerű programokban is így van. A legnagyobb közös osztós programunk esetén adja magát, hogy a számítás érdemes különválasztani. Ezt a változatot fogjuk áttekinteni.

Először lássuk a pszeudokódot!

LNKO 2/szétzedett program:

Kérj be két pozitív egész számot!

Számítsd ki a lnko-t!

Írd ki osztót!

VÉGE.

A program így jóval egyszerűbb lett: a felhasználóval való kommunikáció lett a fő feladata, a számítás egyetlen utasításra redukálódott.

legnagyobb függvény:

bemenet: két pozitív egész

kimenet: egy pozitív egész

lnko=1, osztó=2

Ha a második kisebb, mint az első

Cseréld fel a két számot!

Amíg az első szám nagyobb vagy egyenlő az osztóval

Amíg mindkét szám osztható osztóval

Mindkét számot oszd el osztóval!

lnko-t szorozd meg osztóval!

Növekedd osztót eggyel!

Add vissza osztót!

VÉGE.

A függvény viszont **nem kommunikál a felhasználóval**. (Ez a legtöbb függvény esetén igaz!) Elvégzi a legnagyobb közös osztó kiszámítását, és a kiszámított értéket visszaadja a hívónak.

Most lássuk a programot magát!

```

/*****
***** //LNKO program *****/
*****
#include <stdio.h>
/*****

int legnagyobb(); // deklaráció
int legnagyobb(int,int); // prototípus
// csak az egyik kell

/*****
int main(void){ // definíció
/*****

    int szam1,szam2,x;

    printf("Adj meg 2 poz. egeszt: ");
    scanf("%d%d",&szam1,&szam2);

    x=legnagyobb(szam1,szam2);

    printf("*****");
    printf("\nLNKO = %d\n*****\n",x);
    return 0;
}

```

```

/*****
int legnagyobb(int szam1, int szam2){
/* A legnagyobb közös osztót számolja */
// definíció
/*****

    int osztó=2,lnko=1;
    if(szam1>szam2){ // csere jön
        int temp=szam1;
        szam1 = szam2;
        szam2 = temp;
    }
    while( szam1>=osztó ){
        while(szam1%osztó==0
        && szam2%osztó==0){
            szam1 = szam1 / osztó;
            szam2 = szam2 / osztó;
            lnko = lnko * osztó;
        }
        osztó = osztó + 1;
    }
    return lnko;
}

```

A program futásának eredménye, ha a felhasználó 30-at és 20-at ad meg:

```

Adj meg 2 poz. egeszt: 30 20
*****
LNKO = 10
*****

```

A forráskód magyarázata

Megjegyzések ► A C nyelv két megjegyzéstípust támogat.

/* */ megjegyzések: Ez a típus többsoros is lehet. Kezdetét a /* jelzi, innentől kezdve bármit írunk, azt a fordító nem veszi figyelembe egészen a */ karakterpárig. A /* */ pár ránézésre hasonlít a zárójelezéshez, viszont nem az, mert **nem ágyazható egybe** két megjegyzés.

Zárójelezésnél megtehetjük, hogy ezt írjuk: $y=(a+b*(c+d))*2$, a megjegyzésnél viszont /* X /* Y /* Z /* : itt Z /* már nem része a megjegyzésnek! Azaz mindig az első /* a megjegyzés vége.

Ez akkor kellemetlen, ha szeretnénk a kód egy részét megjegyzésbe dugni, mondjuk hibakeresési célból, és van megjegyzés ebben a részben. A dolgot orvosolhatjuk a // megjegyzések alkalmazásával.

// megjegyzések: a sor végéig tartanak. A C++ nyelvből utólag került át a C-be, ezért a **régebbi** szabvány szerint működő **fordítóprogramok nem támogatják**. Ahogy a példaprogram második sorában látszik, benne lehet a /**/ megjegyzés belsejében.

#include <stdio.h> ► A # kezdetű sorok az előfeldolgozó számára adnak utasítást. A #include azt kéri, hogy az előfeldolgozó szűrje be a programkód szövegébe, ennek a sornak a helyére, a <> között megadott nevű fájlt. Jelen esetben az stdio.h fájl beszurását kérjük. Ha a fájl nevét <> között adjuk meg, akkor a fordító ezt a fájlt azokban a mappákban keresi, amelyek be vannak állítva a számára.

Visual C++ 2003 esetén, alapértelmezett beállításokkal történő telepítéskor pl. a

c:\Program Files\Microsoft Visual Studio .NET 2003\vc7\include\

mappa ilyen. Itt megtaláljuk az stdio.h fájlt is. Az előre beállított include mappák sorát kibővíthetjük, további mappákat adhatunk hozzá. Visual C++-ban a Solution Explorerben jobb klikk a projekten → Properties → C/C++ → General → Additional Include Directories listát tudjuk bővíteni.

Ha nem egy rendszer fejlécfájl szeretnénk beszerkeszteni, hanem sajátot, amelyet a programunkkal együtt tárolunk, azt idézőjelek között kell megadnunk, így: #include "sajat.h"

stdio.h: A Standard Input and Output rövidítése, tehát nem stúdió. A .h kiterjesztésű **fejlécállományok (header fájlok)** függvények prototípusait és konstansokat tartalmaznak. Az stdio.h a C nyelv azon szabványos függvényeinek prototípusát tartalmazza, melyek az adatok beolvasását és kiírását végzik. A képernyőre írás és billentyűzetről olvasás mellett ide tartozik a fájlkezelés is, erről később lesz szó.

A C nyelv szabványos függvénykönyvtára adott téma köré csoportosítva kerültek különböző fejlécállományokba. Gyakran fogjuk használni az stdlib.h-t, ami a Standard Library rövidítése, és sokféle függvényt tartalmaz, pl. memóriakezelést, processzkezelést, stb. A math.h a matematikai függvényeket tartalmazza, pl. gyökvonás, szögfüggvények, stb., a string.h szövegkezeléssel, a time.h időkezeléssel foglalkozik, és így tovább.

Típus ► A könnyebb feldolgozhatóság érdekében az azonos jellegű adatokat egyformán kezeljük, és típust rendelünk hozzájuk. A számítógép memóriájában minden adat bitek sorozata. Egy adat típusa azt jelenti, hogy az adatot leíró bitek/bájtok sorozatát **hogyan kell értelmezni**. Minden típusra meghatározott, hogy mely műveletek értelmezhetők rá és mi az értékészletük.

A C-ben az adatok típusait a következő csoportokba osztjuk:

- egyszerű típusok
 - egész típusok (többféle előjeles és előjel nélküli)
 - valós típusok (többféle pontossággal)
 - karakter
 - felsorolás
- származtatott típusok
 - mutató
 - tömb
 - struktúra
 - unió
 - bitmező

Azonosítók ► A változók, konstansok, függvények, foglalt szavak (kulcsszavak) nevei azonosítók. A C nyelvben az azonosítók

- az angol ABC kis- és nagybetűiből,
- számokból és
- aláhúzás karakterekből (_)

állhatnak, szóköz vagy más karakter nem lehet bennük, és magyar magánhangzókat sem használhatunk. Számmal nem kezdődhetnek. A C nyelv **érzékeny a betűméretre (case sensitive)**, ezért pl. a nev, Nev, NEV, nEv azonosítók különbözőek. Foglalt szó neve nem adható más azonosítónak,

Pl. nem jó változónevek:
int int, while, állat, 4ever;

Pl. jó változónevek:
int a, Macska, NEM, x81, _baba, ___, nagy_ember;

Pl. jó, de kerülendő változónevek:
int o, l, While;

Azért nem gépi kódban programozunk, mert érteni akarjuk a programot. Az azonosítók megválasztásánál is ezt kell szem előtt tartani, ezért nem javasoltak a fenti azonosítók. Az 0 a nullával, az 1 az 1-gyel téveszthető össze könnyen, a While hasonlít egy foglalt szóra. (Természetesen nem csak a While, hanem a többi foglalt szóra hasonlító azonosító is kerülendő. Pl. nem túl olvasható ez a kód: while (While<while) ..., ahol a harmadik azonosítóban egy egyes számjegy található...)

Foglalt szavak (kulcsszavak) ► Olyan fenntartott szavak, amelyek használatát szintaktikai szabályok határozzák meg. Ezek olyan szavak, melyek nem adhatók változónévnek, függvénynévnek, stb.

A C kulcsszavai: auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while

Függvények ► Ahogy láttuk, az algoritmus egyes részeit érdemes kiemelni külön függvénybe, hogy ezáltal áttekinthetőbb kódot kapjunk, és az ismétlődő funkciókat csak egyszer kelljen megvalósítani. A C nyelvben maga a főprogram is függvénybe kerül, ez a **main függvény**. A main függvény paramétere **void**, ami azt jelenti, hogy hiányzik. A void szót elhagyhatjuk, írhatunk egyszerűen int main(){...}-t is.

A C programok elindításakor mindig a main függvény indul el, függetlenül attól, hogy az hol található a kódban, van-e írva elé más függvény, vagy éppen melyik forrásfájlban található.

A függvény függvényfejből és függvénytörzsből épül fel.

A **függvényfej** megadja a **függvény típusát** (milyen típusú adatot ad vissza), a **függvény nevet**, valamint **paramétereinek típusát és nevét**.

A **függvénytörzs** tartalmazza a függvényt felépítő utasítássorozatot, beleértve más függvények meghívását is.

A **függvénydefiníció** maga a függvény, azaz a függvényfej és a függvénytörzs együtt.

A **függvény prototípusa** a majdnem ugyanaz, mint a függvényfej pontosvesszővel lezárva, de még a paraméterek nevét sem kell megadni. A példában: int legnagyobb(int,int);

A **függvénydeklaráció** a függvény prototípusa paraméterek nélkül. Csak a függvény típusát és nevét tartalmazza, üres paraméterlistát adunk. A példában: int legnagyobb();

A **prototípus és deklaráció jelentősége**, hogy egy C program csak olyan függvényt tud meghívni, amelynek deklarációja megelőzi a kódban a függvényhívást. A C szabvány azt ajánlja, hogy ne a deklaráció, hanem az annál több információval bíró **prototípus előzze meg a hívás helyét**. Nem kell deklaráció vagy prototípus abban az esetben, ha a hívott függvény definíciója (megvalósítása) megelőzi a hívás helyét. Gyakran írjuk meg úgy a programot, hogy a main függvény a végére kerül, éppen azért, hogy ne kelljen külön leírni a prototípust.

A függvény definíciója másik C fájlban is lehet, mint ahol használjuk (meghívjuk) a függvényt. Ilyenkor az egyes **C fájlok külön is fordíthatók**, tehát csak azt a fájlt kell újra lefordítani, amelyet megváltoztattuk, így gyorsabban létrejön a futtatható állomány. (Visual C++-ban a Build parancs csak a megváltoztatott fájlokat fordítja újból, míg a Rebuild az összeset.) A külön lefordított fájlokban található függvényekből **függvénykönyvtár**at lehet létrehozni, melyek jellemzően .lib kiterjesztésű fájlokba kerülnek. Ilyen függvénykönyvtárakban található a szabványos függvények, mint a printf, scanf és a többi. Függvénykönyvtár létrehozását a 21. fejezetben fogjuk látni.

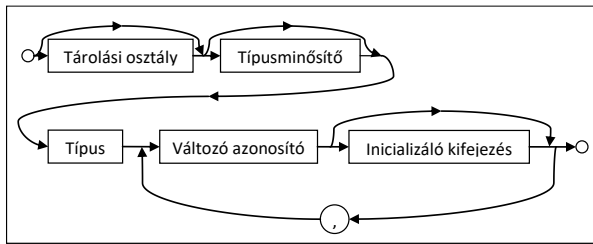
A külön lefordított, akár függvénykönyvtárban, akár object fájlban lévő függvényeket a fordítóprogram **linker** része szerkeszti egybe.

A legtöbb függvényt úgy használjuk, ahogy a matematikai függvényeket: kap valamilyen paramétert, esetleg többet is, és visszaad egy eredményt, pl. y=sin(x). A fenti programban a **legnagyobb** függvény ehhez hasonlóan működik: két egész számot kap paraméterként, és visszaadja az eredményt a **return** utasítással. Tehát a **main** függvényben x-be az az érték kerül, ami a **legnagyobb** függvény return utasítása után írt **lno** változóban van.

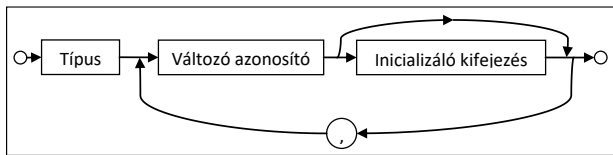
A main függvényben is látunk egy return-t, ami 0-t ad vissza. Ezt a 0-t az operációs rendszer, vagy a programot elindító egyéb más program kapja meg. A program által visszaadott értéket nagyon ritkán használja fel más program. Az operációs rendszernek visszaadott érték sajnos ellentétes a C nyelv logikájával, a C-ben ugyanis a 0 jelenti a HAMIS-at, azaz a rosszat, míg az operá-

ciós rendszernek akkor adunk vissza 0-t, ha a program hibátlanul befejeződött.

A változódefiníció ► szintaxis diagramja:



A tárolási osztály (pl. static) és a típusminősítő (pl. const) megadása nem kötelező, ezekről a 20. fejezetben lesz majd szó. Az eddigi példákban és a továbbiakban az egyszerűsített változódefiníciót használjuk:



```
int oszto=2, lnko=1;
```

Típus: int
Változó azonosító: a, b, c
Inicializáló kifejezés: =0

A listában a változók sorrendje tetszőleges. Egy a lényeg: a változó legyen definiálva, mielőtt először használjuk.

A C99 előtti szabványokban változót definiálni csak a blokk elején lehet, azaz a { után állhatnak változódefiníciók, melyek más utasításokkal nem szakíthatók meg. pl.:

```
{
    int a;
    a=0; /* Változódefiníciók között nem lehet más utasítás! */
    int b=0, c;
    ...
}
```

Ez a kód a C99 előtti szabványok szerint hibás, az a=0; csak a definíciókat követheti. A C++ nyelv azonban ezt megengedi, és később a C99 szabvány is átvette. Ha kódunknak fordulni kell régi szabványt támogató fordítóval is, ezt a megoldást kerülnünk!

Kifejezés ► minden, aminek értéke van (konstans, változó, összetett kifejezés, függvényhívás).

Operátor: műveleti utasítás, műveleti jel. A C-ben sok ilyen van, pl. +, -, *, /, %, &&, ||, <<, ++, =, ==, <, stb.

C-ben igen gazdag operátorkészlet található, így a legtöbb dolog megoldható kifejezés utasítással.

Például:

- `c = a + b;` : ez a kifejezés utasítás összeadja *a*-t és *b*-t, az összeget pedig *c*-be helyezi.
- `a - b;` : ez is egy kifejezés, bár nincs semmi értelme. Kiszámítja a és b különbségét, de az eredménnyel nem csinál semmit. A C-ben egy kifejezés értéke "eldobható".

printf ► Ez a függvény nagyon sokat tud, most megismerkedünk néhány funkciójával. Később, a 12. fejezetben majd alaposan körülbjárjuk a kiíró és beolvasó függvényeket.

Két speciális vezérlőkaraktert használhatunk a formátum-sztringben, ezek a \ és a %.

A \ segítségével olyan karaktereket írhatunk ki, amelyeket egyébként nem tudnánk.

A \n újsor jelet jelent, azaz a \n utáni szöveg új sorba kerül. Ez a jel azért is fontos, mert a printf-fel kiírt sor után, ha ismét kiírunk valamit, az nem kerül automatikusan új sorba, ehhez be kell írunk a \n-t. A példaprogramban két sor csillag közé írtuk az eredményt.

A \t egy tabulátor karaktert ír ki.

A \" egy idézőjelet, a \' egy aposztrófot ír ki. A \\ pedig maga a \ karakter. Vigyázni kell tehát a \ használatára, mert ha pl. egy fájl elérési útvonalát akarjuk kiírni, és ezt írjuk: printf("C:\nagy\nevek.txt");, az eredmény ez lesz:

```
C:
agy
evек.txt
```

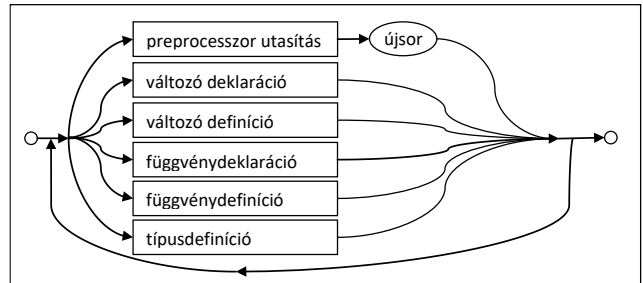
A kívánt eredmény eléréséhez printf("C:\\nagy\\nevek.txt"); szükséges.

A % változók kiírásában segít. Láttuk már, hogy a % egész számot ír ki 10-es számrendszerben, azaz decimálisan. Amikor más adattípusokat tanulunk, megtanuljuk majd, hogy azokhoz milyen betű(k) tartozik/nak. A % ennél többet is tud.

A %5d azt jelenti, hogy a kiírt szám 5 helyet foglal el a képernyőn. Ha a szám ennél rövidebb, akkor szóközöket ír elé. Ez a funkció akkor hasznos, ha egymás alá akarunk számokat írni táblázatszerűen. Ha a szám nem fér el 5 helyen, akkor annyi helyet foglal, amennyin elfér. Ha %05d-t írunk, akkor a szám elé 0-k kerülnek. A %+5d azt jelenti, hogy a szám elé a + előjel is kiíródik, nem csak a -. A %-5d azt jelenti, hogy a szóközök a szám után lesznek, nem előtte, azaz balra igazítódik. A %+5 is helyes. A %-05 esetén a 0-t nem veszi figyelembe a printf, mert a szám után írt nullák elrontanák a számot (elé írva rendben van: 00234, de utána nem jó: 23400). A %% maga a százalékjel.

Ez a felsorolás csak akkor ér valamit, ha kipróbáljuk a gyakorlatban is. Írjon tehát programot ezek kipróbálására!

Egy C program legmagasabb szintű megadása szintaxis diagrammal:



Az egyes részekről később lesz szó.

6. Típusok, változók, konstansok, műveletek

Az előző fejezetekben néhány egyszerű, egész számokat használó algoritmussal találkoztunk. Most megismerkedünk a C nyelv néhány újabb elemével:

- a valós számok tárolására szolgáló `double` típussal,
- a `for` ciklussal
- és néhány új operátorral, azaz műveleti jellel.

6.1 Másodfokú egyenlet gyökei

Matematikából tudjuk, hogy az $ax^2 + bx + c = 0$ egyenlet gyökeit az $x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ képlet adja. Az egyenletnek akkor van valós gyöke, ha a négyzetgyökjel alatt álló diszkrimináns nemnegatív. Írjunk programot, amely bekéri a három paraméter (`a,b,c`) értékét, és kiírja a gyököket, ha léteznek!

Az algoritmus pszeudokódja a következő lesz:

```
Másodfokú egyenlet program
Kérd be a három paramétert!
Számold ki a diszkrimináns!
Ha a diszkrimináns nemnegatív
    Számold ki a két gyököt a megoldóképlettel!
    Írd ki a két gyököt!
Egyébként
    Írj ki hibaüzenetet!
VÉGE.
```

Az algoritmus kódja C nyelven:

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

int main(void) {
    double a,b,c,diszkr,x1,x2;

    printf("Masodfoku egyenlet gyokeinek"
           "\n szamitasa.\n\na = ");
    scanf("%lg",&a);
    printf("b = ");
    scanf("%lg",&b);
    printf("c = ");
    scanf("%lg",&c);

    diszkr = b*b - 4*a*c;

    if( diszkr >= 0.0 ) {
        x1 = (-b + sqrt(diszkr)) / (2*a);
        x2 = (-b - sqrt(diszkr)) / (2*a);
        printf("\nA gyokok:\n\n"
               "\nx1 = %g\nx2 = %g\n",x1,x2);
    }
    else{
        printf("Az egyenletnek nincsenek"
               "\n valos gyokei.\n");
    }
    return 0;
}
```

A programot `gcc`-vel fordítva linkelni kell a matematikai könyvtárat `-lm` kapcsolóval.

A program futásának eredménye:

Masodfoku egyenlet gyokeinek szamitasa.

```
a = 1
b = 6
c = 5
```

A gyokok:

```
x1 = -1
x2 = -5
```

Magyarázat ► A programban ki kell számítanunk a diszkrimináns négyzetgyökét. Ehhez az `sqrt` függvényt használjuk, melynek prototípusa a `math.h` fejlécállományban található, ezért ezt is beszerkesztjük. Mindegy, hogy a beszerkesztésnél az `stdio.h`, vagy a `math.h` van-e előbb.

A program összes változója `double` típusú lesz. A valós számokat a számítógép lebegőpontos formátumban tárolja, ezért a `double`-t **lebegőpontos típusnak** is nevezik. A különböző adattípusok számítógépes megvalósításával a 10. fejezetben foglalkozunk, most csak röviden: a lebegő pont a tizedes pontra utal, amit az angolszász országokban (és a számítástechnikában) a tizedesvessző helyett használnak. Tízest számrendszerben például a $842 \cdot 10^{-2}$ -ként, a 0.0456 -ot $4.56 \cdot 10^{-2}$ -ként adhatjuk meg lebegőpontosan: a tizedespontra az első értékes számjegy után kerül, és az egészet tíz megfelelő hatványával szorozzuk. A számítógép kettes számrendszert használ, ott természetesen kettes számrendszerben történik a szorzás.

A `double` típus beolvasása és kiírása az `int`-hez hasonlóan a `scanf` és `printf` függvényekkel történik, `%` után megadva a megfelelő típusazonosítót. A `double` típus az egyetlen olyan típus a C-ben, ahol a `scanf` és a `printf` eltérő típusazonosítót használ: a **`scanf %lg-t`**, a **`printf %g-t`**. Ha valaki bizonytalan, írjon mindkét helyre `%lg-t`, mert ezt a fordítók el szokták fogadni, bár nem szabványos, fordítva viszont, azaz a `scanf`-be `%g-t` írva hibásan fog működni a program!



A diszkrimináns számításánál figyeljük meg, hogy a négyzetre emelés szorzással valósítjuk meg. **A négyzet és a köb számítását mindig szorzással végezzük!** Egyéb (akár nem egész) hatványok számítására a `math.h`-ban található `pow` függvény megfelelő.

A matematikai képletekben a **szorzást** gyakran nem szoktuk **jelölni**, a programban azonban **kötelező!**

Figyeljük meg, hogy a programban `b*b-4*a*c` szerepel, vagyis nem kellett zárójeloznunk így: `b*b-(4*a*c)`, azaz a C, a matematikához hasonlóan, csak az „azonos erősségű”, azaz azonos **precedenciájú** műveleteket értékeli ki balról jobbra. **A szorzás precedenciája nagyobb, mint a kivonásé**, ezért előbb értékeli ki a szorzás eredménye, aztán a kivonásé. Ha ez nem felel meg, zárójeloznunk!

A használt műveletek sorrendje precedencia szerint (a legerősebb fent):

- + és - mint előjel
- *, / és % szorzás és osztás jellegű művelet
- +, - mint összeadás jellegű művelet
- <, >, <=, >= relációs műveletek
- ==, != relációs műveletek
- && logikai ÉS
- || logikai VAGY
- = értékadás

A `diszkr = b*b - 4*a*c` művelet esetén tehát a sorrend:

1. `b*b`
2. `4*a`
3. `(4*a)*c`
4. `(b*b)-((4*a)*c)`
5. `diszkr=((b*b)-((4*a)*c))`

A példaprogramban csak oda került zárójel, ahová kell. Ha programot ír, és bizonytalan, nyugodtan használjon zárójelket!

Szövegek két sorban. A könyv kéthasábos szedése miatt nem fértek el a hosszú szövegek egy sorban. A C nyelv lehetővé teszi, hogy az idézőjelbe írt szövegeket, azaz a **sztring konstansokat** több részre, így akár több sorba osszuk úgy, hogy a két (vagy több) részszoveg mindegyikét "-ek közé zárjuk, és a részek közé csak whitespace karaktereket (szóköz, újsor, stb.) írunk. Ezeket a szövegeket az előfeldolgozó automatikusan összefűzi.

Konstansok ► Eddigi programjainkban is számos esetben találkoztunk konstansokkal. A szövegkonstansok (**sztring konstansok**) megadása idézőjelek között történik. Pl. "Hello World!\n".

A karakterek tárolására alkalmas char adattípusról még nem beszélünk. A sztring karakterek sorozata, melynek végét egy speciális karakter jelzi. A karakterek sorban egymás mellett helyezkednek el a memóriában, azaz tömbben. Erről később szólnunk részletesebben. A **karakter konstansokat** aposztrófok között adjuk meg: 'A', '@', stb. A speciális karakterek megadása a \ segítségével történik, így pl. '\n', '\t', '\\', stb. Ezek egy karakternek számítanak. Csak ezeknél fordulhat elő, hogy az aposztrófok közé egynél több karaktert írunk, de itt is csak azért, mert ezek egynek számítanak. **FIGYELEM!** A **sztring konstans** és a **karakter konstans** **ne keverjük össze**, mert bár ránézésre mindössze annyi a különbség, hogy idézőjelek vagy aposztrófok között vannak, egészen máshogy kezeli ezeket a C fordító! Tehát pl. az "A" és 'A' nem helyettesíthető egymással!

Az **int típusú egész konstansok** írásmódjában nincs semmi meglepő: számjegyek sorozata, melyet előjel egészíthet ki. Például: -5, 0, 1686, stb.

Az egész számokat megadhatjuk nyolcas és tizenhatos számrendszerben. Ha a **konstans nyolcas (oktális) számrendszerben** adjuk meg, 0-val kell kezdeni, pl. 012 (= tíz), 050 (= negyven), stb. **Tizenhatos (hexadecimális) számrendszerben megadott konstans** kezdete 0x vagy 0X. Például: 0x21 (= harminchárom), 0X10 (= tizenhat). 0xa2 (= százhatvan-kettő).

A **double típusú lebegőpontos konstansok** a számjegyeken kívül vagy pontot, vagy e vagy E betűt tartalmaznak. Például: 1.2, -4.0, 3e2, -4.1E-002, 91.4e+1, stb. Az e és E jelentése azonos, tíz hatványával való szorzást jelent, azaz pl. 3e2 jelentése 300.0, -4.1E-002 jelentése -0.041, 91.4e+1 jelentése 914.0. Az e ill. E után csak egész szám állhat. Double konstans pl. a 431. (ponttal a végén) és a .23 (ami 0.23-at jelent) is.

Vigyázzunk a használat során! Mennyi double $x=3/4$?
 x-be 0.0 kerül

Mert 3 is és 4 is egész, és először a hányadosukat számolja ki a számítógép. Az egész osztás eredménye egész szám, amit a számítógép nem kerekít, hanem levágja a tört részt, azaz 0.75 helyett 0 van az = jobb oldalán.

A számítás akkor fog 0.75-os eredményt adni, ha a két érték közül legalább egy lebegőpontos, azaz pl. $x=3.0/4.0$, vagy $x=3.0/4$, stb.

A jelenség nem csak a konstansokat, hanem a változókat is érinti. Például `int a=3,b=4; double d=a/b;` Itt ugyanúgy 0.0-t kapunk. A megoldás ekkor az, hogy **explicit** (közvetlen) **típuskonverziót** alkalmazunk, vagyis legalább az egyik egész változó elé odaírjuk zárójelben, hogy double. Pl.: `x=(double)a/b;`, vagy `x=a/(double)b;` A típuskonverzió precedenciája magasabb, mint az osztásé, ezért nem szükséges így zárójelezni: `((double)a)/b.` A zárójelbe írt típusnevet egyúttal típuskonverziós operátornak nevezzük.

6.2 Átlag

Írjunk programot, amely előre ismeretlen mennyiségű valós számokat kér a felhasználótól, és kiírja a számok átlagát! A számsorozat végét a 0.0 érték jelzi.

N darab szám átlagát így számoljuk ki:

$$\text{átlag} = \frac{\text{szám1} + \text{szám2} + \dots + \text{számN}}{N}$$

Azaz összeadjuk a számokat és elosztjuk az összeget a darabszámmal.

A megoldás algoritmus:

Átlag program

Jegyezd meg, hogy a darabszám és az összeg nulla!

Kérj a felhasználótól egy számot!

Amíg a szám nem nulla

Növelj összeget a számmal!

Növelj darabszámot eggyel!

Kérj a felhasználótól egy számot!

Ha a darabszám nagyobb nullánál

Írd ki az összeget és a darabszám hányadosát!

Egyébként

Írd ki, hogy nincs átlag!

VÉGE.

Mint látjuk, nincs szükség arra, hogy az összes számot külön-külön megjegyezzük, elég az összegüket.

A program C nyelven:

```
#include <stdio.h>

int main(void) {
    double szam, osszeg=0.0;
    int N=0;

    printf("Adj egy szamot (vege:0): ");
    scanf("%lg", &szam);
    while (szam != 0.0) {
        osszeg += szam;
        N++;
        printf("Adj egy szamot (vege:0): ");
        scanf("%lg", &szam);
    }
    if (N>0)
        printf("Atlag: %g\n", osszeg/N);
    else printf("Nincs atlag.");
    return 0;
}
```

Miután megnézte a megoldást, próbálja meg önállóan megírni a programot! Ugye, hogy nem is olyan egyszerű?

Új operátorok ► A while ciklusban két új operátorral találkozunk: a +=-vel és a ++-szal. Ezek a rövidebb kód írását segítik.

Az `osszeg += szam;` jelentése: `osszeg = osszeg + szam;`, vagyis az `osszeg` változóban található értéket növelem meg `szam`-mal. Minden alapműveletre létezik ilyen operátor: +=, -=, *=, /=, %=.

Az `N++;` jelentése `N = N + 1;`, vagyis `N` értékét növeli eggyel. A ++ operátor csak egészekre definiált, valósokon nem (fordítóprogramtól függ, hogy valóban működik-e). Ez a sor természetesen így is írható: `N += 1;`. Ha `N` értékét eggyel csökkenteni akarjuk, `N--;`-t írhatunk. A ++-t inkrement, a --t dekrement operátornak is nevezzük.

Nézze át a korábbi példákat, és cserélje ki az új operátorokra a régieket, ahol lehet!

6.3 Fordítva

Írjuk ki az összes olyan háromjegyű számot, amely előlről hátra ugyanaz, mint hátulról előre! Például: 121, 707, 444, stb.

Ezeknek a számoknak a formátuma ABA. Mivel három jegyű számokról van szó, A értéke nem lehet nulla, 1-től 9-ig mehet. B értéke 0-9-ig mehet.

Kétféle algoritmus ötletét vetjük fel:

1. Brute-force módszer: 100-tól 999-ig megyünk egy ciklusban, és minden számról eldöntjük, hogy megfelel-e az elvárásoknak. Az egyes számjegyeket a maradékképzés és az osztás operátorokkal nyerhetjük ki: ha i a ciklusváltozónk, akkor $i/100$ adja az első számjegyet (egész osztás eredménye egész, nincs kerekítés, csonkolás történik, így pl. $199/100=1$), $i\%10$ pedig az utolsó számjegyet adja. Az algoritmus pszeudokódjának és C nyelvű megvalósításának elkészítését az olvasóra bízunk.
2. Két egymásba ágyazott ciklust használunk, ahol B értékét adja a belső, A értékét a külső.

A második algoritmus pszeudokódja a következő lehet:

Fordítva program

```
Legyen A=1!  
Amíg A kisebb 10-nél  
  Legyen B=0!  
  Amíg B kisebb 10-nél  
    Írd ki ABA-t!  
    Növeld B-t egygel!  
  Növeld A-t egygel!
```

VÉGE.

A program C nyelven:

```
#include <stdio.h>  
  
int main(void) {  
    int A,B;  
    A=1;  
    while (A<10) {  
        B=0;  
        while (B<10) {  
            printf("%d ",A*100+B*10+A);  
            B++;  
        }  
        A++;  
    }  
    return 0;  
}
```

A `printf("%d ",A*100+B*10+A);` helyett írhattuk volna, hogy `printf("%d%d%d ",A,B,A);`.

Nagyon gyakori, hogy egy algoritmusban egy ciklusváltozót léptetünk valamettől valameddig. Már korábban is találkoztunk ilyen programokkal. Sok programozási nyelv, így a C is rendelkezik olyan ciklusutasítással, amelyik kifejezetten ilyen ciklusok létrehozására szolgál: ez a **for ciklus**.

A C for ciklusa a következő felépítésű:

```
for( kezdeti_beállítás ; feltétel ; léptetés ){  
    utasítások_a_ciklus_magjában;  
}
```

Ez a következőképp nézne ki while ciklussal:

kezdeti_beállítás;

```
while(feltétel){  
    utasítások_a_ciklus_magjában;  
    léptetés;  
}
```

A két megoldás teljesen azonos egymással, bármikor helyettesíthető egyik a másikkal.

Például:

```
for(i=0; i<10; i++)printf("%d",i);
```

Ez a ciklus 0-tól 9-ig írja ki a számokat. (Ha i eléri 10-et, $i<10$ hamissá válik, már nem fut le újból a ciklus magja.)

A Fordítva program C nyelvű megvalósítása for ciklusokkal:

```
#include <stdio.h>  
  
int main(void) {  
    int A,B;  
    for (A=1; A<10; A++){  
        for (B=0; B<10; B++){  
            printf("%d ",A*100+B*10+A);  
        }  
    }  
    return 0;  
}
```

A for ciklusra is igaz, hogy ha csak egy utasítás van a ciklus magjában, akkor a kapcsos zárójelek elhagyhatók. A fenti programban ez az állítás mindkét ciklusra igaz. (A külsőre is, mert abban csak egy másik for ciklus van, melynek része a printf.) A tovább rövidített program ez lesz:

```
#include <stdio.h>  
  
int main(void) {  
    int A,B;  
    for (A=1; A<10; A++){  
        for (B=0; B<10; B++){  
            printf("%d ",A*100+B*10+A);  
        }  
    }  
    return 0;  
}
```

Aki bizonytalan abban, hogy mi egy és mi nem egy utasítás, inkább zárójelezzon, abból sosem lehet baj!

6.4 Vektor elforgatása

Írjunk programot, amely bekéri a felhasználótól egy kétdimenziós valós értékű vektor koordinátáit, valamint egy szög fokban, és kiírja a képernyőre annak a vektornak a koordinátáit, amely az eredeti vektor adott szöggel történő elforgatásának eredménye!

Az elforgatott koordinátákat a következő képletek adják:

$$\begin{aligned}x_{uj} &= x_{régi} \cdot \cos\alpha - y_{régi} \cdot \sin\alpha \\ y_{uj} &= x_{régi} \cdot \sin\alpha + y_{régi} \cdot \cos\alpha\end{aligned}$$

A megvalósításnál figyelniünk kell arra, hogy a *math.h*-ban definiált szögfüggvények nem fokban, hanem radiánban várják a szögeket, ezért a felhasználótól kapott szöget radiánba kell konvertálnunk a konverzió képlete:

$$\alpha_{rad} = \alpha_{fok} \cdot \pi / 180$$

A program C-ben így néz ki:

```

#include <stdio.h>
#define _USE_MATH_DEFINES // a VC++-nak kell
#include <math.h>

int main(void) {
    double xregi, yregi, xuj, yuj, alfa;

    printf("x="); scanf("%lg", &xregi);
    printf("y="); scanf("%lg", &yregi);
    printf("alfa="); scanf("%lg", &alfa);

    alfa = alfa * M_PI / 180.0;

    xuj = xregi * cos(alfa) - yregi * sin(alfa);
    yuj = xregi * sin(alfa) + yregi * cos(alfa);

    printf("\nxuj=%g\nyuj=%g\n", xuj, yuj);

    return 0;
}

```

A program második sorában látjuk a `#define _USE_MATH_DEFINES` kifejezést, vagyis definiáljuk a `_USE_MATH_DEFINES` szimbólumot (lásd a 31. fejezetben). Ez azért szükséges, hogy elérjük a `math.h`-ban definiált `M_PI` konstans, vagyis a π értékét. A szimbólum definiálása meg kell előzze a `#include <math.h>`-t. Ha gcc-vel fordítjuk, ne feledjük a matematikai könyvtárat hozzálinkelni `-lm` kapcsolóval!

Gondolkozzunk együtt!

G6.1 Írjon C programot, amely bekér a felhasználótól egy pozitív egész számot, és kiírja a szám számjegyeinek összegét! Pl. 1981 $\rightarrow 1+9+8+1 = 19$.

Ha önállóan nem megy, segítségül az algoritmus:

```

Jegyösszegző program
Az összeg legyen nulla!
Kérj be egy számot!
Amíg a szám nem nulla
    A szám tízzel való osztási maradékát add az
    összeghez!
    Oszd el a számot tízzel!
Írd ki az összeget!
VÉGE.

```

G6.2 Írjon C programot, amely bekér a felhasználótól három valós számot, és eldönti, hogy a három szám lehet-e egy háromszög oldala!

G6.3 Írjon egy teljes C programot, amely megkeresi és a szabványos kimenetre írja azt a legnagyobb háromjegyű számot, amelynek számjegyösszege megegyezik prímtényezőinek összegével. (pl. $378=2\cdot3\cdot3\cdot3\cdot7$, ezek összege 18, ami egyenlő a jegyek összegével is.)

G6.4 Írjon olyan algoritmust, amely egy valós szám első két tizedesjegyét egy egész típusú változóba teszi. Ügyeljen arra, hogy a valós szám negatív is lehet, a kinyert két tizedesjegy viszont nemnegatív alakú legyen! Például be: 123.456, ki: 45. Be: -0.0987, ki: 9. Be: 3.0, ki: 0.

G6.5 Írjon C programot, amely folyamatosan kér a felhasználótól valós számokat egész addig, amíg az 0.0-t nem ad! Írja ki a felhasználó által adott értékek közül a legnagyobbat! A legnagyobb meghatározásakor ne vegye figyelembe az utoljára megadott 0.0-t!

Tipp: az első számot külön kell beolvasni, mert ez lesz a maximumot tároló változó kezdeti értéke. A maximumot tároló változónak tilos hasra ütéssel kezdőértéket adni „legyen mondjuk -1.0e100, mert annál

úgyis biztos nagyobb lesz a maximum” módszerrel, mert mi van akkor, ha a felhasználó csak ennél kisebb értéket ad meg? Ez súlyos elvi hiba, mindig igazi értékkel inicializáljunk!

Oldja meg egyedül!

L6.1 Írjon C programot, amely kiírja a táblára a 12x12-es szorzótáblát! A tábla így kezdődik:

	1	2	3	4	...
1	1	2	3	4	...
2	2	4	6	8	...
3	3	6	9	12	...
4	4	8	12	16	...
...					

L6.2 Írjon C programot, amely bekér a felhasználótól egy egész számot, és kiírja, hogy a szám tökéletes-e! Tökéletes szám az, amely megegyezik osztóinak összegével, az osztók közé értve 1-et, de a számot magát nem. Például $28=1+2+4+7+14$.

Ha önállóan nem megy, segítségül az algoritmus:

```

Tökéletes számok program
Az összeg legyen nulla, a számláló meg egy!
Kérj be egy egész számot!
Amíg a számláló nem nagyobb, mint a szám fele
    Ha a számláló a szám osztója
        Add számlálót az összeghez!
        Növeld számlálót eggyel!
    Ha a szám egyezik az összeggel
        Írd ki, hogy tökéletes!
    Egyébként
        Írd ki, hogy nem tökéletes!
VÉGE.

```

L6.3 Írjon C programot, amely bekér a felhasználótól két valós számot, és kiírja, hogy barátságosak-e! Két szám barátságos, ha az egyik szám osztóinak összege a másikat adja, a másik osztóinak összege pedig az egyiket, az osztók közé értve 1-et, de a számot magát nem. Például: 220 és 284 barátságos, mert 220 osztói: $1+2+4+5+10+11+20+22+44+55+110=284$ 284 osztói: $1+2+4+71+142=220$.

L6.4 Írjon C programot, amely valós számokból álló koordináta-párokot kér be. Addig olvassa a koordináta-párokat, amíg a felhasználó mindkét koordinátaként nullát nem ad. Írja ki, hogy a koordináták hány százaléka esett az egység sugarú körön belülre!

L6.5 Írjon C programot, amely bekér egy pozitív egész számot a felhasználótól, és a számot fordítva írja ki a képernyőre! Pl.: Be 12345, ki: 54321. Be 12000, Ki: 00021

7. A C utasításai

Az előző fejezetekben megismerkedtünk a C utasításainak többségével. Most áttekintjük az összeset, mivel nincs belőlük túl sok.

A következő táblázat tartalmazza a C nyelv összes utasítását:

Utasítás	Formátum
Üres utasítás	; vagy {}
Összetett utasítás (utasításblokk)	{ utasítás1; utasítás2; ... utasításN; }
Tárolási egységek (változók, konstansok, függvények) definíciója / deklarációja	típus változó1[=kezdőérték] [, változó2...];
Kifejezés utasítás	Operátorok, függvényhívások, konstansok és változók sorozata, ;-vel lezárva.
Feltételes elágazás	if(feltétel)utasítás; if(feltétel)utasítás1; else utasítás2; switch(egész típusú érték){ case érték1: utasítások; case érték2: utasítások; ... case értékN: utasítások; default: utasítások; }
Ciklus	while(feltétel)utasítás; for(inicializálás; feltétel; léptetés)utasítás; do utasítás while(feltétel);
Ugró utasítások*	return; return kifejezés; break;* continue;* goto címke;*

*A ciklusban használt break, a continue és a goto utasítások elkerülésével általában áttekinthetőbb, karbantarthatóbb kód szokott keletkezni, éppen ezért lehetőség szerint nem használják őket.

Üres utasítás ► Akkor használjuk, ha valahová szükséges utasítást írni, de nem akarunk.

A ;;; vagy a {} {} {} négy üres utasítás.

Például a következő kódrészlet meghatározza azt a legkisebb N-et, amire igaz, hogy $szam \leq N$, ahol $N = 3^n$.

```
for (N=1; szam>N; N*=3);
```

A ciklus magja üres utasítás.

Összetett utasítás ► Akkor használjuk, ha egynél több utasítást akarunk olyan helyen elhelyezni, ahol csak egy utasítás állhat (lásd: if, else, for, while magja). Továbbá, ha egyéb okból össze akarunk zárni utasításokat. **A függvények törzsét kötelező blokkba tenni.**

Minden összetett utasítás elején lehet változót definiálni, a változó a blokk végéig él. Például:

```
{ int x=5, i;  
  for(i=0; i<10; i++)x+=i;  
  printf("%d\n", x);  
}
```

~~==10~~ // Fordítási hiba: x nem létezik a blokkon kívül

Tárolási egységek definíciója/deklarációja ► A C nyelv a program részeit egységesen, **tárolási egységekben** kezeli. Tárterület-foglaló tárolási egységek az adatok (konstansok, változók), kódgeneráló tárolási egységek a függvények. A tárolási egységet a programban vagy külső modulban definiálni kell, használatukat a kódban meg kell előznie a deklarációjuk, ami lehet maga a definíció is. A tárolási egységeknek típusa van: adatok esetén: az adatokat reprezentáló bitek/bájtok sorozatát hogyan kell értelmezni (egész, lebegőpontos, karakter, stb.), függvények esetén: van-e visszatérési értékük, és ha van, azt mint adatot, hogyan kell értelmezni. (void: ha nincs visszatérési érték)

A változódefinícióról az 5.2. fejezetben volt szó.

Például:

```
int a;  
double d=2.1;  
char alma='a', c=alma, x;
```

Kifejezés utasítás ► Az 5.2. fejezetben beszéltünk a kifejezésekről. Emlékeztetőül: Kifejezés minden, aminek értéke van (konstans, változó, összetett kifejezés, függvényhívás). A kifejezés utasítás egy kifejezés pontosvesszővel lezárva.

példa:

```
a=1; <= ez is kifejezés utasítás, lásd értékadás operátor
```

```
1; <= szintaktikusan helyes, de értelme nincs
```

```
a+2; <= szintaktikusan helyes, de értelme nincs
```

```
b=a+2;
```

```
x=(-b+sqrt(b*b-4.0*a*c))/(2.0*a); <= összetett kifejezés
```

A C-ben egy kifejezés értéke "eldobható", ahogy a fenti példákban is láttuk. Ez azt jelenti, hogy egy függvény által visszatartott értéket sem kell felhasználnunk. Például nyugodtan leírogatjuk egy programban, hogy

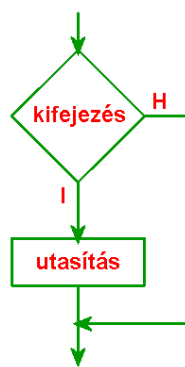
```
sqrt(81.0);
```

Sok értelme nincs, hiszen a négyzetgyököt nem használtuk fel. Van olyan függvény azonban, ahol van értelme. Például ilyen a printf, amely az általa kiírt karakterek számát adja vissza, így a következő utasítás értelmes:

```
int n=printf("Ez vajon hany karakter?");
```

Feltételes elágazás ► Jól ismerjük már az if illetve if-else utasításokat, újdonság viszont a switch. Tekintsük át folyamatábra segítségével az utasítások működését!

if :

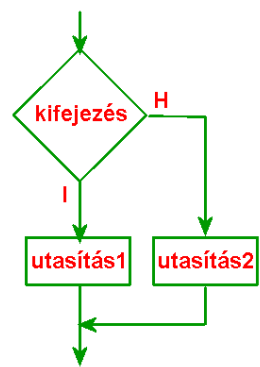


Forma:

```
if(kifejezés)utasítás;
```

```
if(kifejezés)utasítás1;  
else utasítás2;
```

if - else :



Ha a zárójelek között logikai IGAZ értékű kifejezés áll, akkor végrehajtódik az if utasítása. Ha hamis, az else utasítása hajtódik

végre. Ha nincs else, akkor egyből az if-et követő utasítással folytatódik a program.

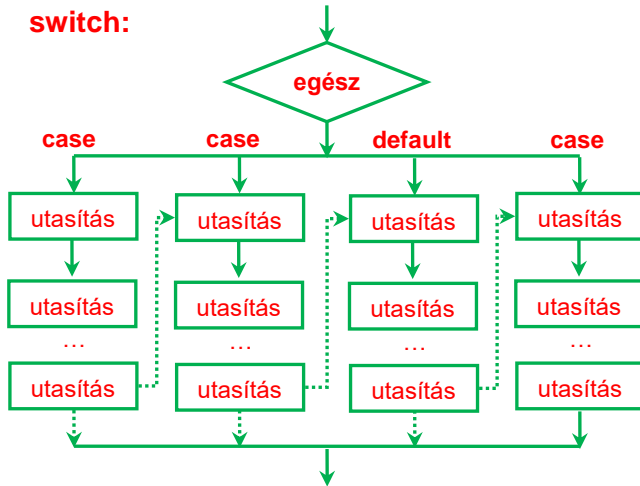
Példák:

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

Az if és az else között nem lehet más utasítás, csak az if magjában lévő utasítás, vagy összetett utasítás.

A switch utasítás egy egész típusú érték alapján dönti el, melyik ágon folytassa a program végrehajtását.

switch:



Forma:

```
switch(egész_érték){
    case konstans1: utasítás; utasítás; ...utasítás;
    case konstans2: utasítás; utasítás; ...utasítás;
    ...
    case konstansN: utasítás; utasítás; ...utasítás;
    default: utasítás; utasítás; ...utasítás;
    case konstansN+1: utasítás; utasítás; ...utasítás;
    ...
    case konstansM: utasítás; utasítás; ...utasítás;
}
```

A switch összehasonlítja a paraméterként kapott kifejezés értékét a case-ek után írt konstanssal, és ha megegyezik valamelyikkel, akkor az utána megadott utasításokkal folytatódik a program végrehajtása. Ha egyik konstanssal sem egyezik meg, akkor a default után írt utasításokkal folytatódik a működés (olyan, mint az else az if-nél).

A switch az a kivétel az utasítások között, amelynél **nem csak egy utasítás**, vagy blokk állhat a case vagy a **default** után, hanem bármennyi.

Ha az adott case ág utasításait végrehajtotta a program, akkor nem az egész switch utáni utasítással folytatódik a program futása, ahogy pl. az if-else-nél, hanem a következő case vagy default utáni utasításon. Ha azt akarjuk, hogy ne fusson rá a program a következő case-re vagy a defaultra, akkor **break** utasítással ki kell lépni a switchből.



Bármennyi case és egy darab default lehet, ezek sorrendje tetszőleges. Lehet a default az első, az utolsó (ez a leggyakoribb), vagy bármelyik közbenső.

A default elhagyható, hasonlóan az if-else-hez.

Nem kötelező külön sorba írni a case-eket/defaultot, de az áttekinthető kód érdekében erősen javasolt.

Példák:

```
int n;
char ch;
...
switch(n){
    case 1: printf("egy\n"); break;
    case 100: n++;
    case 101: printf("szazegy\n"); break;
    default: printf("valami\n");
}
```

Ha *n* értéke 1, kiírjuk, hogy egy. Ha *n* értéke 100, akkor megnöveljük eggyel, és kiírjuk, hogy „szazegy”. Ha *n* értéke 101, akkor kiírjuk, hogy „szazegy”. Ha egyik sem igaz, kiírjuk, hogy „valami”.

```
switch(n){
    default:
        printf("Alapértelmezett.\n"); break;
    case 2:
        printf("Kiírta, hogy 2, ");
        n += 1;
    case 3:
        printf("és azt is, hogy 3.\n");
}
```

Most a defaultot az elejére írtuk.

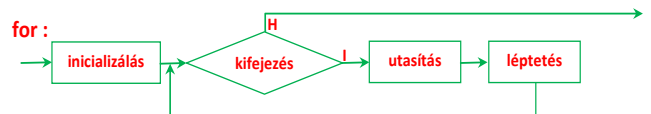
```
scanf("%c",&ch);
switch(ch){
    case 'i':
    case 'I':
        printf("Igen");
        break;
    case 'n':
    case 'N':
        printf("Nem");
        break;
    default:
        printf("Nemigen");
}
```

A case utáni részt üresen is hagyhatjuk.

A karaktereket (betűk, számjegyek, írásjelek) **char** típusú változóknak tároljuk. A char típus is egész típusú, mert minden karakter egy egész számnak felel meg. Erről részletesebben a 12. fejezetben lesz szó. A sztring (szöveg) karakterekből épül fel, de nem egész típusú, mert nem egy egész érték alkotja, hanem sok, ezért **sztringgel nem lehet a switch-et használni!** A char értékek scanf-fel történő beolvasásához és printf-fel történő kiírásakor a %c formátumazonosítót használjuk. Nem egyedi karakterek, hanem szövegek beolvasására és kiírására mást fogunk használni.

Ciklusok ▶ A három ciklusfajta közül kettővel már találkoztunk, a while-lal és a for-ral. A harmadik, a do-while abban tér el az előző kettőtől, hogy **hátral tesztelő**, azaz először lefut a ciklusmag, majd ellenőrzi a ciklusfeltétel teljesülését az utasítás, és ha az IGAZ, ismét végrehajtja a ciklusmagot. A különbség annyi tehát, hogy hátral tesztelő ciklusoknál a ciklusmag legalább egyszer lefut. A while és a for ciklus **elől tesztelő**.

Nézzük a for ciklust:



Például:

Szám felezése:

```
for(i=100; i>0; i/=2)printf("%d ",i);
```

Összetett kifejezés is használható mindhárom részben. Például írjuk ki 2 első 20 hatványát!

```
int h, n;
for (h=2, n=1; n<=20; n++, h*=2)
    printf("%d ", h);
```

Az elválasztásra a **vessző operátort** használtuk. Két változó kapott induló értéket, és mindkettőt változtattuk a léptetésben.

A vesszőt a C nyelv kifejezésekben operátorként használja, de a függvényhívásoknál használt vessző nem vessző operátor, azaz a printf("%d",a); függvényhívásnál a vessző nem operátor. Amikor operátor, akkor a többi műveleti jelhez hasonlóan használhatjuk, tehát értelmes az x=(3,4); kifejezés utasítás, x-be 4 kerül: a vessző a két operandusa közül a jobb oldalit adja vissza. Ebben az esetben muszáj zárójelezni, mert minden operátor közül a vessző precedenciája a legkisebb. x=3,4; is értelmes szintaktikailag: x-be 3 kerül, a 4 meg csak úgy van, nem használjuk semmire. Az x=(3,4)-hez hasonló értékadást általában nem szoktuk használni, a vesszőt elsősorban a for ciklusban való elválasztásra találták ki.

A következő példában a feltétel és a léptetés összetett kifejezés összetett. Két utasítást tettünk a for belsejébe, ezért ezeket {} közé kell tenni, azaz összetett utasításként adjuk meg. A for inicializáló kifejezését üresen hagytuk.

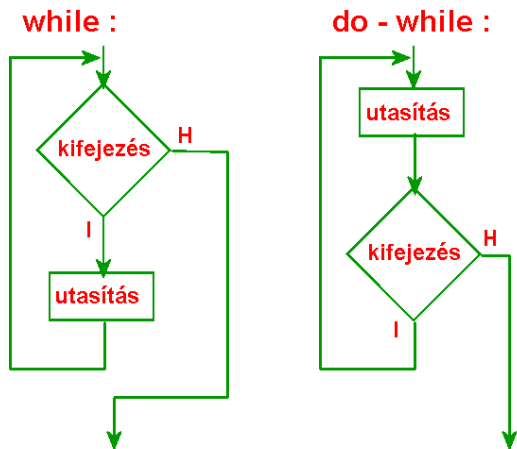
```
double x, y;
...
for (; x<100.0 && y>-2.2; x+=1.0, y-=x) {
    printf("x erteke: %g\n", x);
    printf("y erteke: %g\n", y);
}
```

Nem csak az inicializáló kifejezés, hanem a **for fejének három része közül bármelyik elhagyható.**

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

Ahol a feltétel üresen maradt, ott a feltétel mindig IGAZ-nak számít, ami azt jelenti, hogy **végtelen ciklust** kaptunk. Végtelen ciklusból pl. a return-nel ki lehet lépni, bár ez a kód áttekinthetőségét rontja.

A while és a do-while ciklus folyamatábrája a következő:



Ha az utasításból másolatot készítünk a while ciklus elé, akkor a do-while működését kapjuk. Ha pedig a do-while ciklust egy if belsejébe tesszük, ahol az if feltétele ugyanaz, mint a ciklus feltétele, a while működését kapjuk. → Bármelyik ciklus hasz-

nálható bárhol. Azt a ciklust célszerű használni, amelyik a legáttekinthetőbb kódot eredményezi az adott algoritmusnál.

Példa: írjuk ki az 1000-nél kisebb prímszámokat! Nézzük meg mindhárom ciklusfajttával a megoldást!

Az algoritmus a következő: 2-től 1000-ig léptetünk egy változót egy ciklusban, és minden lépésben megvizsgáljuk, hogy a változó értéke prím-e. Ha prím, kiírjuk. A vizsgálatot ugyancsak egy ciklussal végezzük, így egymásba ágyazott ciklusokat használunk.

Az elől tesztelő ciklusokat használó algoritmus pszeudokódjának elkészítését az olvasóra bizzuk.

For ciklussal:

```
#include <stdio.h>

int main(void) {
    int i;

    for (i=2; i<1000; i++) {
        int j, prim=1;
        for (j=2; j<i && prim==1; j++)
            if (i%j==0) prim=0;
        if (prim==1) printf("%3d ", i);
    }
    return 0;
}
```

While ciklussal:

```
#include <stdio.h>

int main(void) {
    int i=2;

    while (i<1000) {
        int j=2, prim=1;
        while (j<i && prim==1) {
            if (i%j==0) prim=0;
            j++;
        }
        if (prim==1) printf("%3d ", i);
        i++;
    }
    return 0;
}
```

Do-while ciklussal:

```
#include <stdio.h>

int main(void) {
    int i=2;

    do {
        int j=2, prim=1;
        if (j<i) {
            do {
                if (i%j==0) prim=0;
                j++;
            } while (j<i && prim==1);
        }
        if (prim==1) printf("%3d ", i);
        i++;
    } while (i<1000);
    return 0;
}
```

A külső ciklusnál nem kellett mást tennünk, mint az elejéről a végére vinni a while-t, és a while helyére beírni, hogy do. Ez azért lehetséges, mert $i=2$ -re biztosan lefut a ciklus. A belső ciklus azonban nem futna le $i=2$ esetén, mert a $j<i$, azaz $2<2$ feltétel nem teljesül. Ha ugyanúgy járnánk el, mint a külső ciklusnál, akkor az $i\%j==0$ igaz volna, mert $2\%2$ az nulla, hiszen minden szám osztható magával. Ezért is nem $j<=i$ -ig ment a belső ciklus. Ezek azok az apróságok, amelyekre gondolni kell programírás közben, és ami csak sok gyakorlással válik természetessé.

A do-while megoldásba tehát bekerült egy plusz if, ami, ha végiggondoljuk, mindössze egyetlen egyszer játszik szerepet, az $i=2$ -nél. Minden más esetben a while $j<i$ feltétele megakadályozza az önmagával való osztás vizsgálatát. Ha először kiírjuk, hogy 2, utána pedig a külső ciklus 3-tól indul, akkor nem szükséges az if, ami majdnem ezerszer futna le feleslegesen.

Ha már optimalizálásról beszélünk, nézzük meg, mit tehetünk még? Ha 3-ról indítjuk a prímeresést, akkor felesleges a páros számokat vizsgálni, azok úgysem prímek. Legyen tehát az $i++$ helyett $i+=2$, így i értéke 3, 5, 7, 9, ... lesz! További gyorsulást érhetünk el, ha a belső ciklus nem i -ig, hanem i négyzetgyökéig megy: ha egy számnak nincs osztója a négyzetgyökéig, akkor az prim. (Miért? Gondolja végig!) Lehetne írni a belső ciklusba, hogy $j<=\text{sqrt}(i)$, de ez azzal a következménnyel járna, hogy a belső ciklus minden iterációban kiszámolja ugyanannak az i -nek a négyzetgyökét. Számoljuk ki tehát a gyököt a ciklus előtt!

Vegyük észre, hogy már az eredeti algoritmus is tartalmaz optimalizációt: a belső ciklus vizsgálja, hogy $\text{prim}==1$, így ha összetett számot találunk, egyből befejeződik. Az algoritmus tökéletesen működne, ha csak $j<i$ lenne a ciklusfeltétel, csak lassabb lenne.

Íme, az optimalizált algoritmus:

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

int main(void) {
    int i=3;

    printf("2 ");
    do{
        int j=2, prim=1, gyok;
        gyok=(int)(sqrt(i)+0.5);
        do{
            if(i%j==0) prim=0;
            j++;
        }while( j<=gyok && prim==1 );
        if(prim==1)printf("%3d ", i);
        i+=2;
    }while(i<1000);
    return 0;
}
```

Még néhány trükk: a gyököt int-ként tároljuk, mert a $j<=$ gyok művelet gyorsabb, ha mindkettő egész, és az algoritmust átgondolva megtehetjük ezt.

$\text{gyok}=(\text{int})(\text{sqrt}(i)+0.5)$; utsítást írtunk. Miért?

Az sqrt függvény double típusú paramétert vár és double típusú eredményt ad. A C automatikusan alakít át egész adattípusból lebegőpontosba, vissza viszont nem, hiszen adatvesztés lehet a következmény. (Például 4.7-ből 4 lesz, azaz elvész a szám tört része, amit nem tudunk visszanyerni. Ha 4-et alakítunk 4.0-vá, akkor nyilván nem volt adatvesztés.) Az automatikus átalakítást **implicit típuskonverzió** nevezzük. Az (int) odairásával jelezzük, hogy tisztában vagyunk az adatvesztéssel, mégis kérjük az átalakítást, ez az **explicit típuskonverzió**, amiről már volt szó.

Miért adunk hozzá 0.5-et? Mert kerekíteni szeretnénk. Lebegőpontos számításokkor erre figyeljünk! Lebegőpontos számítások közben kvantálási és kerekítési hibák léphetnek fel. Ennek eredménye lehet, hogy bár például a pontos eredmény 4.0 lenne,

3.9999999999-et kapunk, és (int)3.999999999 az bizony 3, és nem 4! Szóval lebegőpontos számításoknál mindig gondoljunk a kerekítési hibákra! Erről lesz még szó a lebegőpontos számábrázolás kapcsán a 10. fejezetben.

Ugró utasítások ► Ezeket csak nagyon óvatosan szabad használni, mert könnyen áttekinthetetlen, módosíthatatlan programkódhoz juthatunk, kilógnak a strukturált programozás eszköztárából. A break utasítás használata elkerülhetetlen a switch utasításban, a return-t muszáj használni az értéket visszaadó függvényekben. A break és continue használata ciklusokban, a goto-é pedig mindenhol kerülendő. A returnnel is bánjunk óvatosan!

A figyelmeztetések után nézzük röviden, mire is jók az egyes ugró utasítások!

return: függvényekből való kilépésre használjuk.

Például egy egész szám abszolút értékét számító függvény:

```
int abszolut(int x) {
    if(x<0) return -x;
    return x;
}
```

A függvény használata pl.: $\text{int a}=\text{abszolut}(-5)$; A -5 helyett változó is lehet.

Két return is szerepel a függvényben. Ha x negatív, a függvény $-x$ -et ad vissza, és azonnal ki is lép, tehát a return x ; már nem hajtódik végre. Éppen ezért nem kellett odairni, hogy else return x ; (Így is lehet, de szükségtelen.)

A fenti függvényt egyszerűbben megvalósíthatunk volna a **feltételes operátor**ral is: A feltételes operátor az

```
if(feltétel) y=valami1;
else y=valami2;
```

típusú értékadások egyszerűsítésére szolgál. Különlegessége, hogy ez az egyetlen háromoperandusú művelet a C-ben. A feltételes operátor a kérdőjel-kettőspont, azaz $?:$.

Szintaxis: *feltétel ? kifejezés1 : kifejezés2*

Ha a feltétel IGAZ, az operátor *kifejezés1* értékét adja eredményül; ha HAMIS, a *kifejezés2* értékét adja eredményül.

Például: $\text{max} = \text{a}>\text{b} ? \text{a} : \text{b}$;

A *max*-ba *a* és *b* közül a nagyobb kerül. A változók egészek, valóságok, karakterek egyaránt lehetnek.

Az abszolút függvény feltételes operátorral:

```
int abszolut(int x) {
    return x<0 ? -x : x;
}
```

break, continue: a két utasítás a három ciklusutasításon belül használható, ezen kívül a break még a switch-ben is működik, ahogy ezt a switch-nél láttuk.

A break segítségével kiugorhatunk a ciklusból. Például:

```
for(i=0; i<1000; i++){
    printf("i=%d\n", i);
    if(i==20)break;
    printf("Jok vagyunk!\n");
}
```

Itt ránézésre van egy ciklusunk, amely 0-tól 999-ig megy el, valójában azonban, ha a ciklusváltozó eléri a 20-at, kilépünk a ciklusból. Break helyett alkalmazunk megfelelő ciklusfeltételt! Ha a ciklusfeltétel nagyon bonyolult lenne, hozzunk létre egy segédváltozót, és azt vizsgáljuk a ciklusban.

Például:

```
int seged=1;
while (seged==1) {
    if (a>b) seged=0;
    i++;
    if (i==n) seged=0;
}
```

Alakítsa át az előző példát break nélkülivé (nincs szükség segédváltozóra)!

A continue segítségével átugorhatjuk a ciklushátralevő részét. Például:

```
for (i=0; i<1000; i++){
    printf("i=%d\n", i);
    if (i==20) continue;
    printf("Jok vagyunk!\n");
}
```

While és do-while ciklus esetén a feltételre ugrunk, for esetén pedig a léptetésre. A példában 20 esetén nem írjuk ki, hogy „Jok vagyunk!”.

Ha több ciklus van egymásba ágyazva, akkor a break/continue csak a közvetlenül befoglaló ciklusra hat, tehát nem ugrik ki az összesből. A switch-ben lévő break is csak a switch-ből ugrik ki.

goto: Ezzel az utasítással függvényen belül bárhová ugorhatunk, függvények között azonban nem lehet átugrani.

Például:

```
#include <stdio.h>

int main(void) {
    int i=0;
start:
    printf("%d ", i);
    i++;
    if (i<20) goto start;
    return 0;
}
```

A címke C azonosító, azaz az angol ABC kis- és nagybetűiből, számokból és aláhúzás jelekből állhat, számmal nem kezdődhet. A címkét nem kötelező a sor elejére írni, de így legalább látszik. A ciklusokat a számítógépek valójában a fentihez hasonló módon valósítják meg. A fenti kód for ciklussal:

```
int main(void) {
    int i=0;
    for (i=0; i<20; i++) printf("%d ", i);
    return 0;
}
```

Az összes ugró utasítás közül a goto használata kerülendő leginkább, mert ezzel kuszálható leginkább össze a kód.

Gondolkozzunk együtt!

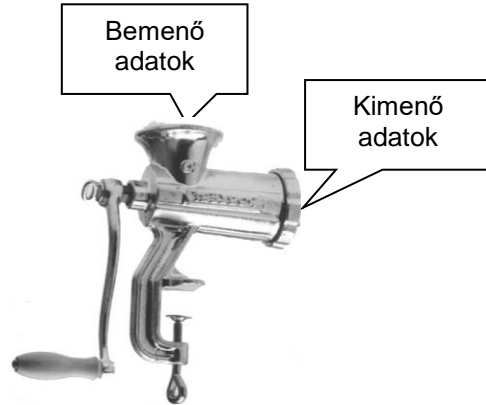
Oldja meg egyedül!

8. Egyszerű függvények

A program részekre bontásának, azaz a **programszegmentálás** legfontosabb eszközei a függvények. Két fő funkciójuk:

1. A többször használt programrészeket csak egyszer kell leködölni.
2. A bonyolult algoritmusok szétbontva egyszerű részekre áttekinthetővé válnak.

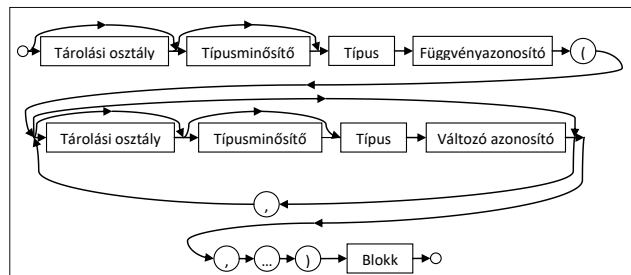
A függvények általános felépítése:



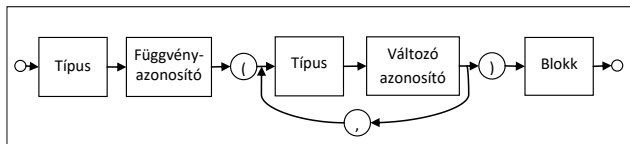
A függvényt tekinthetjük tehát egy adatfeldolgozó egységnek, amely kap bemenő adatokat, és ezeket felhasználva produkál valamilyen eredményt. Vannak függvények, amelyek nem kapnak bemenő adatot (például ha szükségünk van egy véletlenszámra, meghívjuk a rand() függvényt), kimenő adatot viszont valamilyen formában mindegyik szolgáltat (nem tudunk mondani olyan értelmes példát, amely nem ad eredményt). A kimenő adat általában valamilyen érték illetve adatszerkezet, néha a képernyőn megjelenő vagy fájlba kerülő információ.

A C nyelvben egy függvény **tetszőleges számú bemenő paraméterrel** és **egy darab visszatérési értékkel** rendelkezhet. Ha egynél több visszatérési értéket kell adni, azt trükkkel oldjuk meg, erről a 17. fejezetben lesz szó bővebben.

A függvénydefiníció részei szintaxis diagrammal:



Kétszer is látjuk a tárolási osztály – típusminősítő – típus – azonosító sorozatot. Ez nem véletlen, hiszen korábban volt róla szó, hogy a függvény és a változó egyaránt tárolási egység. A fenti definícióban a tárolási osztály, típusminősítő és a ... opcionálisan használható, és ritkábban is használjuk ezeket, most a legfontosabb részekkel foglalkozunk, melyek az alábbi szintaxis diagramon láthatók. A ...-ről azért annyit jegyezzünk meg, hogy ez teszi lehetővé a printf és scanf függvény számára, hogy a formátumsztring után bármennyi változót megadhassunk. Általában nem élünk ezzel a lehetőséggel, azaz a függvényeknek **pontosan annyi paramétert** kell adni híváskor (aktuális paraméterek), amennyi a függvénydefinícióban feltüntetünk (formális paraméterek).



Például ▶ írjunk függvényt, amely paraméterként kap két egész számot, és visszaadja a számok összegét!

```
int osszead(int a, int b) { // formális paraméterek
    return a+b;
}
```

Vagy, ha valaki külön változót szeretne létrehozni az eredménynek (nem szükséges):

```
int osszead(int a, int b) {
    int c;
    c=a+b;
    return c;
}
```

A függvény aktiválása (meghívása) például a következőképpen történhet:

```
int main(void) {
    int x, y, z;
    printf("3+2=%d\n", osszead(3, 2));
    x=5;
    y=3;
    z=osszead(x, y); // aktuális paraméterek
    printf("%d+%d=%d\n", x, y, z);
    return 0;
}
```

Az egyik függvényben definiált változók nem láthatók a másik függvényben. Például ha $a=3$; szerepelne a `main()`-ben, a fordító hibát jelezne, hiszen az `a` az `osszead` függvényben van definiálva. Ennek a következménye az is, hogy a két függvényben lehetnek azonos nevű változók, ezek között semmiféle kapcsolat nem lesz.

A függvény fejlécében szereplő `int a`, `int b` ránézésre olyan, mint egy változódefiníció. Nem csak ránézésre olyan, ez **változódefiníció!** Az egyszerű változódefinícióval (`int c`;), vagy `int x,y,z`; szemben az a különbség, hogy a **függvényfejben** minden egyes változó elé **ki kell írni a típust**, tehát az `int a,b` hibás megoldás lenne. Az `int a`, `int b` és az `int c` közötti legfontosabb különbség, hogy az `a` és `b` azért van a `()` között, mert számukra **kezdőértéket** kell adnia a függvény hívójának. Azaz a **függvény meghívása nem más, mint kezdőérték-adás a paraméterlistán definiált változóknak.**

A kezdőérték-adáson kívül további kapcsolat nincs a függvény hívásakor adott aktuális paraméterek és a paraméterlistán definiált változók, a formális paraméterek között. Vagyis a következő kódrészletben a `main()` függvény `a` és `b` változó eredeti értékét fogja kiírni, a függvény nem hat vissza a hívás helyére.

```
#include <stdio.h>
void nemcserel(int a, int b) {
    int c=a; a=b; b=c;
}
int main(void) {
    int a=3, b=4;
    nemcserel(a, b);
    printf("a=%d, b=%d\n", a, b);
    // eredmény: a=3, b=4
    return 0;
}
```

A `nemcserel()` függvény **visszatérési típusa void**, azaz hiányzik, tehát nem ad vissza semmit. Az ilyen függvényekbe **return nem szükséges, de használhatjuk return; formában.**

Betű-e a karakter? ▶ Írjunk függvényt, amely a paraméterként kapott karakterről eldönti, hogy az betű-e, vagy bármi más (pl. szám, írásjel)! A függvény adjon vissza egy egész számot, melynek értéke 1, ha betűt kapott, és 0, ha számot! (Vagyis 1 jelezze az IGAZ-at és 0 a HAMIS-at!)

Tudnivalók: a feladat szövegében nem szerepel, de ilyenkor mindig az angol ABC betűire gondolunk, mert a magyar karakterek figyelembe vétele nagyon megbonyolítaná a megoldást. Az angol ABC betűi ugyanis egymás melletti számokkal vannak kódolva. Erről bővebben a 12. fejezetben beszélünk. A kis- és nagybetűket más-más számok jelzik.

Figyelem! **A kis- és nagybetűk nem szomszédosak** egymással. Valahogy így helyezkednek el egymáshoz képest:

<egyéb karakterek>... 'A', 'B', 'C', ..., 'Y', 'Z'... <egyéb karakterek>..., 'a', 'b', 'c', ..., 'y', 'z'... <egyéb karakterek>.

Megoldás: ellenőrizzük, hogy a vizsgált karakter benne van-e az ['A','Z'] vagy az ['a','z'] zárt intervallumban. A függvény C kódja, valamint a kipróbálásához írt program:

```
#include <stdio.h>
int betu_e(char ch) {
    if((ch>='A' && ch<='Z') ||
        (ch>='a' && ch<='z'))
        return 1;
    return 0;
}
int main(void) {
    char ch='4';
    if(betu_e(ch)==1)
        printf("%c betu.\n", ch);
    else printf("%c nem betu.\n", ch);
    ch='w';
    if(betu_e(ch)) // egész szám mint logikai érték
        printf("%c betu.\n", ch);
    else printf("%c nem betu.\n", ch);
    return 0;
}
```

A függvényben nem írtunk `else-t` a `return 0`; elé, mert ha teljesült az `if` feltétele, a `return 1`; már kilépett a függvényből.

A `main()` függvény egyértelmű, csak a `betu_e()` függvény második meghívása jelent újdonságot: nem írtunk relációs operátort a visszatérési érték vizsgálatára. En nem hibás. A C-ben az egész számok használhatók logikai értéként. A 0 logikai HAMIS, a nem 0 (tehát bármely egész szám, amely nem 0) pedig logikai IGAZ. A fenti kód tehát helyesen működik.

Érdekes ellentmondás, hogy a sikeresen lefutó `main()` függvény 0-t ad vissza, ami a C-ben HAMIS-at jelent.

Betűk szűrése ▶ Írjunk függvényt, amely a szabványos bemenetről olvas sorvége jelig, és a beolvasott szöveget úgy írja a szabványos kimenetre, hogy kiszűr belőle mindent, ami nem betű, azaz csak a betűket írja ki! Pl. be: „A '48-as forradalom 1848-ban volt.” Ki.: Aasforradalombanvolt”

A **szabványos bemenet** normál esetben a billentyűzet, a **szabványos kimenet** pedig a képernyő.

Megoldás: olvassuk be a karaktereket egy ciklusban, amíg nem újsor jelet kapunk! Minden karakterre ellenőrizzük, hogy betű-e, és ha igen, írjuk ki! A megoldásban felhasználjuk az előző példában látott függvényt is.

Írja meg az algoritmus pszeudokódját!

A C nyelvű megvalósítás:

```
#include <stdio.h>

int betu_e(char ch) {
    if((ch>='A' && ch<='Z') ||
        (ch>='a' && ch<='z'))
        return 1;
    return 0;
}

void szuro(void) {
    char ch;
    do{
        scanf("%c", &ch);
        if(betu_e(ch)) printf("%c", ch);
    }
    while(ch!='\n');
}

int main(void) {
    szuro();
    return 0;
}
```

Ha kipróbáljuk a programot, láthatjuk, hogy a szűrt szöveg csak azután jelenik meg, hogy leütöttük a sor végén az ENTER-t. Ennek az az oka, hogy a scanf csak akkor kapja meg a begépelte szöveget feldolgozásra, amikor a felhasználó leütötte az ENTER-t, és ez nem függ a beolvasandó adatok típusától. A C nyelvben nincs olyan szabványos beolvasó függvény, amely lehetővé tenné a leütött billentyűk azonnali beolvasását.

Módosítsa úgy az algoritmust, hogy a számjegyeket szűrje ki a szövegből! Hozzá kell nyúlni a *szuro()* függvényhez?

Gondolkozzunk együtt!

Oldja meg egyedül!

II. RÉSZ: ISMERKEDÉS A C NYELVVEL

9. Tömbök

Gyakran van szükség nagyobb mennyiségű adat feldolgozására. Az esetek egy részében elkerülhető, hogy az adatokat eltároljuk, ilyen volt az átlagszámító program a 6.2. fejezetben. Más esetekben viszont muszáj megjegyeznünk az adatokat. Nézzük a következő egyszerű feladatot!

Feladat ▶ Kérjünk a felhasználótól 100 darab egész számot, és írjuk ki a képernyőre fordított sorrendben!

Nem tudunk olyan trükköt kitalálni, hogy a számokat ne kelljen valamilyen módon eltárolni. Az eddig tanultak alapján száz darab egész típusú változóra van szükségünk. Mindegyiket külön definiálni pl. `int a,b,c,d,e,f,g,h,...` módon, vagy akár `int a1,a2,a3,a4,a5,a6,a7,...` módon igen macerás, és ha száz helyett százezer értéket kellene tárolni, akkor egyenesen lehetetlen. Tömb alkalmazásával tudjuk megoldani a problémát.

A **tömb** olyan összetett változó típus, amely a programozó által kívánt mennyiségű, azonos típusú adatot képes tárolni. C-ben a tömb méretét a tömb létrehozásakor szögletes zárójelben adjuk meg:

```
int tomb[100];
```

Ez egy százelemű, egészek tárolására alkalmas tömb, pont megfelelő a feladatunk megoldására. A tömb elemei az indexükkel érhetők el. Az **index** egy 0 és N-1 közötti egész szám, ahol N a tömb mérete (olyan, mint egy vektor a matematikában, csak nem 1-től, hanem 0-tól kezdődik az elemek számozása). Jelen esetben N=100, azaz a tömb elemei `tomb[0]...tomb[99]` közöttiek. Ez összesen 100 darab. Aki nem hiszi, számoljon utána!

Magyarázat: a fordító a tömb neve helyére a tömb kezdőelemének címét helyettesíti be, a keresett elem memóriacímét így számítja ki, ha `[i]`-t keressük: `t_cime+i*(egy_elem_mérete)`.

A feladat megoldása tömb segítségével a következő:

```
#include <stdio.h>

int main(void) {
    int tomb[100], i;
    for (i=0; i<100; i++)
        scanf("%d", &tomb[i]);
    for (i=99; i>=0; i--)
        printf("%d", tomb[i]);
    return 0;
}
```

Azaz az első ciklus 0-tól 99-ig feltölti a tömb elemeit, a második ciklus pedig 99-től 0-ig kiírja őket.

További tulajdonságok ▶ A tömb létrehozásakor a méretet konstans értékkel adjuk meg, változót ne használjunk! A C99-es szabványtól kezdődően a szabvány megengedi, hogy a méretet változóval adjuk meg, azonban ezt a lehetőséget sok fordító nem támogatja, többek között a Visual C++ sem. Ha platformfüggetlen megoldást akarunk készíteni, akkor tömböt csak konstanssal definiálunk. Ha fordítási időben nem ismert, mekkora tömbre lesz szükség, használjuk dinamikus tömböt! Ezekről a 18. fejezetben lesz szó bővebben.

Mindig kötelező a tömbnek méretet adni! C-ben nincs olyan tömb, ami magától megnő, ha nagyobb tömbre van szükség. Az `int tomb[]`; definíció nagyon súlyos hiba. Ha nem ismert előre, hogy hány adatot kell eltárolnunk, akkor az adatokat nem

tömbben, hanem valamilyen dinamikus adatszerkezetben, például láncolt listában vagy bináris fában tároljuk. Ezekről a 24. és a 27. fejezetben lesz szó. Ha előre ismeretlen mennyiségű adatot kell feldolgoznunk, gondoljuk meg, hogy valóban szükség van-e az adatok tárolására! Az átlagszámításnál láttuk, hogy nem mindig van.

A többi típushoz képest (beleértve a még nem tanult típusokat, például a struktúrát is) a C tömböknek van néhány furcsa, gyakran kellemetlen tulajdonsága. A **tömb nem másolható az = operátorral**, azaz pl. az

```
int tomb1[100], tomb2[100];
tomb1=tomb2;
```

értékadás nem működik, a tömbelemeket egyenként kell másolni, így:

```
for (i=0; i<100; i++) tomb1[i]=tomb2[i];
```

Függvény típusa sem lehet tömb, azaz pl. az

```
int [100] fuggveny() {
    int *x[100];
    return *x;
}
```

függvény nem jó. Tömböt visszaadni mégis tud a függvény, de csak trükkösen. A függvények kaphatnak tömböt paraméterként, azonban ezek a változók nem úgy viselkednek, mint a többi más típusúak. A fenti furcsaságok bővebb kifejtésére és magyarázatára a 17. fejezetben kerül majd sor.

Kezdőérték ▶ tömbnek is adható. Az elemek kezdőértékét kapcsos zárójelek között, vesszővel elválasztva soroljuk fel.

Pl.: `double d[3]={-2.1, 0.0, 3.7};`

A kezdőérték megadásának van néhány speciális változata:

- Adhatunk kevesebb értéket, mint a tömb mérete. Pl.: `double d[3]={-2.1, 1.0};`. Ebben az esetben a megadott értékek a tömb elején lesznek, a tömb végén lévő elemekbe pedig nulla kezdőérték kerül (minden bájttól 0 értékű lesz). Figyelem! Ha nem adunk kezdőértéket egyáltalán, akkor a tömb elemei kezdetben memóriaszeméletet tartalmaznak!
- Ha kezdőértéket adunk a tömbnek, a méretet nem kötelező megadni. Pl.: `d[]={-2.1, 0.0, 3.7};`. Ebben az esetben a fordító megszámlálja, hogy hány értéket adtunk meg, és pontosan ekkora tömböt hoz létre. Most sincs szó tehát „akárhány elemű tömbről”, mert olyan nem létezik a C-ben.

A sztringek tárolása ▶ karaktertömbben történik.

A szöveget többek között a `scanf` függvény segítségével tudjuk beolvasni (bővebben lásd a 12. fejezetben). Pl.: `char szoveg[100]; scanf("%s", szoveg);`. (Jegyezzük meg, hogy **sztring esetén nem írunk & jelet a scanf-be**, ez az egyetlen kivétel!) A felhasználó beír egy szöveget, leüti az ENTER-t, és a szöveg a `szoveg` nevű tömbbe kerül (pontosabban a szövegnek az első whitespace-ig tartó része, több szavak beolvasására a `gets` vagy az `fgets` függvény használható, lásd a 12. fejezetben). Kiírhatjuk pl. a `printf("%s\n", szoveg);` módon.

A beolvasás a példában egy százalékos tömbbe történt. A felhasználó azonban valószínűleg nem egy száz karakter hosszúságú szöveget fog megadni. Például ha a vezetéknévét kérdezzük, nagyon sokféle szöveg hossz elképezhető. A tömb mérete nem változik meg attól, hogy beleteszünk valamit, mivel a tömbök mérete a C-ben fix: akkorák, amekkorának létrehozuk őket. A szöveg mérete sem tud alkalmazkodni a tömb méretéhez. Ebből következik, hogy a C nyelvnek valamiképpen tudnia kell, hol van a szöveg vége a tömbön belül. A C nyelv kitalálói két megoldás közül választhattak: eltárolhatták volna a szöveg hosszát egy külön változóban, ők azonban ehelyett a szöveg végét egy speciális karakterrel, a **lezáró nullával** jelezték. A lezáró nullát karakterkonstansként így jelöljük: `'\0'`, ami nem tévesztendő össze a 0 számjeggyel, mint karakterrel, ami egyszerűen '0'. Ennek az a jelentősége, hogy minden szabványos C függvény, mely sztringekkel dolgozik, a lezáró nulláig dolgozza fel a szöveget.

Sztring kezdőérték karaktertömbnek kétféle módon adhatunk. Az első az, amit az előző oldalon láttunk:

```
char tomb[20]={'S','z','i','a',' ',' ',' '\0'};.
```

Vegyük észre, hogy bár a „Szia!” szöveg négy karakter, a sztring **a tömbben eggyel több helyet foglal el, mert a lezáró nullának is ott kell lennie a végén!**

Mivel ez a módszer igen nehézkes, lehetőség van sztring konstanssal való kezdőérték adásra is:

```
char tomb[20]="Szia!";.
```

Ez a megadás teljesen azonos az előzővel, azaz a lezáró nulla is odakerül. Jegyezzük meg, hogy **sztring esetében csak kezdőérték adásakor használható az = operátor!** A sztringmásolásról a 19. fejezetben lesz szó, és ott ismerkedünk meg majd sok más fontos sztringkezelő algoritmusal is.

Többdimenziós tömbök ► Keresztretjvényt szeretnénk készíteni, mondjuk 20×30 kocka méretben (azaz vízszintesen 20, függőlegesen 30). Ehhez összesen 600 karaktert kell tárolnunk. Sok egyforma adat, előre ismert méret → tömb. A betűket **sorfolytonosan** tároljuk, azaz először az első sor betűi egymás mellett, aztán a második sor betűi, és így tovább, vagyis az egymás mellett lévő betűk közt egyet kell lépni, az egymás alatt lévő betűk közt pedig húszat. Az i. sorban és j. oszlopban lévő betű indexe $i*20+j$, ha i és j 0-tól indul. (Ha i és j 1-től indul, $(i-1)*20+(j-1)$ az index.) Azaz

```
char rejvny[600]; rejtveny[i*20+j]='A';
```

A C nyelv azonban biztosít számunkra ennél kényelmesebb megoldást is, mégpedig a többdimenziós tömbök lehetőségét:

```
char rv[30][20]; rv[i][j]='A';
```

Az rv egy kétdimenziós tömb 30 sorral és 20 oszloppal. A tömböt bejárhatjuk és kiírhatjuk a következő kóddal:

```
for(i=0; i<30; i++){
    for(j=0; j<20; j++)
        printf("%c ",rv[i][j]);
    printf("\n");
}
```

Kettőnél több dimenziós tömböt is létrehozhatunk a fentihez hasonló módon.

Az 1. fejezetben beszéltünk a számítógép memóriájáról, és láttuk, hogy az valójában bájtok egydimenziós tömbje. A C fordító a kétdimenziós tömböt valójában egydimenziós, sorfolytonos alakban tárolja, és pontosan úgy számítja ki a kívánt elem helyét, ahogyan azt a *rejtveny* tömb esetén láttuk. A kettőnél több dimenziós tömböket hasonló módon egydimenziós tömbben tárolja.

Kezdőérték többdimenziós tömbnek ► az egydimenzióséhoz hasonlóan adható, pl.:

```
int t[3][4][2]={34,-5,3,20,12,5,-1,0,4,
77,12,-3,-14,3,1,23,75,2,10,24,78,1,2,7};
```

Azonban, az áttekinthetőség érdekében megadhatjuk csoportosítva is:

```
int t[3][4][2]={
{{34,-5},{3,20},{12,5},{-1,0}},
{{4,77},{12,-3},{-14,3},{1,23}},
{{75,2},{10,24},{78,1},{2,7}}};
```

A kezdőértékek többdimenziós esetben is elhagyhatók, ekkor a nem beállított értékek helyére 0 kerül (ha egyáltalán nincs kezdőérték, a tömb elemei nem nullázódnak, memóriaszemét lesz bennük). Például:

```
#include <stdio.h>

int main(void){
    int i,j,k;
    int t[3][4][2]={
        {{34}},
        {{4,77},{12}},
        {{75},{10},{78,1}}};

    for(i=0; i<3; i++){
        for(j=0; j<4; j++){
            for(k=0; k<2; k++){
                printf("%d ",t[i][j][k]);
                printf(", ");
            }
            printf("\n");
        }
    }
    return 0;
}
```

Gondolkozzunk együtt!

másoljuk át egyik tömbből a másikba az átlag alattiakat!

Oldja meg egyedül!

10. Számábrázolás és az egyszerű adattípusok

10.1. Számábrázolás

Egész számok ábrázolása ► Az 1. fejezetben említettük, hogy minden adat, amit a számítógép tárol, bájtok sorozata. Az egész számok és karakterek éppúgy, mint a lebegőpontos számok. Egy program esetén egy konkrét adattípus mindig ugyanannyi memóriát foglal, azaz ugyanannyi (bináris) számjegyből áll, például az int típus PC-n, 32 bites rendszerben, Visual C++-szal fordítva 32 bites, vagyis 4 bajtos (a pontos szabályokat a fejezet későbbi részén találjuk). Például:

```
00000000 00000000 00000000 00000000 = 0
00000000 00000000 00000000 00000001 = 1
00000000 00000000 00000000 00000010 = 2
00000000 00000000 00000000 00000011 = 3
...
01111111 11111111 11111111 11111111 = 2 147 483 647
10000000 00000000 00000000 00000000 = -2 147 483 648
...
11111111 11111111 11111111 11111100 = -4
11111111 11111111 11111111 11111101 = -3
11111111 11111111 11111111 11111110 = -2
11111111 11111111 11111111 11111111 = -1
```

Ezzel az int típussal tehát -2 147 483 648 és 2 147 483 647 közötti egész számokat tudunk tárolni. Ha ennél nagyobb pozitív, vagy kisebb negatív számokkal kell számolnunk, akkor az int típus nem megfelelő, mert nem tudjuk hová írni a további számjegyeket, más típust kell használnunk, már ha létezik egyáltalán alkalmas típus. A C nyelv többféle egész típust is ismer, ezekkel hamarosan megismerkedünk.

A példában látható, hogy negatív számoknál az első bit 1-es, nemnegatívoknál pedig 0. Az azonos értékű csak előjelben különböző számok között azonban ennél jóval nagyobb a különbség. A magyarázat erre a furcsa ábrázolásra a következő: tekintsük a bitmintát egyszerű bináris számnak, és adjuk össze mondjuk -3-at+3-mal!

```
11111111 11111111 11111111 11111101
+ 00000000 00000000 00000000 00000011
-----
1 00000000 00000000 00000000 00000000
```

Az eredmény egy 33 bites szám. Mivel csak 32 bitünk van, egy bitet el kell dobni, ez pedig a **legnagyobb helyértékű bit** lesz (MSB – Most Significant Bit), és a maradék 32 bitet tartjuk meg, ami nem más, mint a nulla. Bármely számok esetén ellenőrizhetjük, sőt, ha pl. -7-hez +10-et adunk, akkor +3 lesz az eredmény, ugyanígy egy **túlsorduló** bittel (-10-hez adva +7-et -3-at kapunk, túlsordulás nélkül). Az egész számok ilyen ábrázolását **kettes komplementes számábrázolásnak** nevezzük. A C nyelv nem írja elő ezt az ábrázolási módot, azonban a ma használatos számítógépek ezt használják.

Minden műveletnél figyeljünk arra, hogy az eredmény ábrázolható legyen a választott adattípussal. **Figyelem!** Tilos úgy ellenőrizni a túlsordulást, hogy `if (a+b>2147483647) printf("Túlsordulás");`, mert ez az állítás mindig hamis lesz, hiszen az összeg típusa is int, a túlsordulás nem csak akkor tűnik el, ha egy int változóba tesszük az összeget, hanem a processzor eleve a túlsordult bit nélküli eredményt adja.



Lebegőpontos számok ábrázolása ► A lebegőpontos számokat az egészekhez hasonlóan fix darabszámú biten ábrázolja a számítógép. A számot ábrázoló biteket két részre osztjuk: alapra (mantissza) és kitevőre (karakterisztika). A számok lebegőpontos alakja: $alap \times 2^{kitevő}$. Az alapot úgy határozza meg a rendszer, hogy értéke az [1,2) balról zárt, jobbról nyílt intervallumba essen (azaz 1 lehet, 2 már nem), a kitevő pedig ebből adódik. Mivel ilyen formán minden szám 1,xxxx alakú, az 1-et nem kell eltárolni. A 0 nem hozható ilyen alakba, erre egy speciális esetet definiáltak.

A C nem rögzíti, hogy konkrétan mi a lebegőpontos számok formátuma, azonban a gyakorlatban általában az IEEE 754 szabványú ábrázolás a legelterjedtebb.

Típus	Előjel [bit]	Kitevő [bit]	Kitevő eltolás	Értékes jegyek száma [bit]	Összesen [bit]
Fél	1	5	15	10	16
Egyszeres	1	8	127	23	32
Dupla	1	11	1023	52	64
Quad	1	15	16383	112	128

A PC processzora ezek közül a az egyszeres és dupla pontosságú szabványt ismeri. A PC-s fordítók általában a float típusnak az egyszeres pontosságú, a double és long double típusnak a dupla pontosságú lebegőpontos típust feleltetik meg.

A kitevő eltolás azt jelenti, hogy 2^n kitevő esetén $n+eltolás$ érték kerül eltárolásra. Az IEEE szabvány szerint tárolt változók esetében az eltoló kitevő a csupa 0 és a csupa 1 értéket nem veheti fel, ezek speciális számok tárolására vannak fenntartva, ezért egyszeres pontosság esetén [-126, 127], dupla pontosság esetén [-1022, 1023] a lehetséges kitevő tartomány (n). Tíz-es számrendszerben az egyszeres pontosságú típussal ábrázolható tartomány kb. 10^{-38} .. 10^{38} , a dupla pontosságúval kb. 10^{-308} .. 10^{308} .

Speciális számok:

- A 0.0-t nem tudjuk addig osztani 2-vel, míg 1 és 2 közé esik, de az sem engedhető meg, hogy egy „0.0-hoz nagyon közel eső nem 0.0 érték” jelenítse meg, hiszen a 0.0 a leggyakrabban használt szám. A 0.0 esetén a kitevő, előjel és az alap egyaránt 0, azaz a 0.0 bitmintája 0x00000000.
- A csupa 1 kitevő, csupa 0 alap a ∞ -t jelenti (INF). Akkor kapjuk ezt, ha a művelet eredményét nem lehet ábrázolni az adott adattípussal, mert túl nagy.
- A csupa 1 kitevő, nem 0 alap jelentése: nem szám (NaN, Not a Number). Egyéb hiba. Pl. INF*0.

Példa: Hogy néz ki a 6,28 egyszeres IEEE alakban?

- $6,28 = 1,57 \times 2^2$
- előjel=0
- kitevő=0000 0010 (2)
- eltoló kitevő: 1000 0001 (129)
- alap: 1,1001 0001 1110 1011 1000 011 (1,57), az egészrészt (1) nem tárolja
- a szám tehát: 0100 0000 1100 1000 1111 0101 1100 0011 = 0x40c8f5c3

Ki is próbálhatjuk az alábbi kóddal (csak akkor működik jól, ha float és unsigned egyforma bitszámú):

```
float f=6.28f;
printf("%x\n",*(unsigned*)&f);
```

A számítási pontatlanság ► Lebegőpontos számok esetében nem szabad elfelejteni, hogy a számolás nem pontos, mert az értékes jegyek száma korlátozott.

- Nézzünk egy példát tízes számrendszerben! Ha 4 értékes jegy pontosságig tudjuk ábrázolni a számokat, és van a következő két számunk: $8,888 \times 10^3$, valamint 1111×10^{-4} , azaz 8888 és 0,1111. Mindkét számot tudtuk ábrázolni. A kettő összege 8888,1111, azaz $8,8881111 \times 10^3$. De az eredménynek is csak 4 értékes jegyét tudjuk tárolni, tehát a kiszámolt eredmény $8,888 \times 10^3$, azaz mintha az összeadás meg sem történt volna.

Pontatlanság példaprogrammal:

```
//*****
#include <stdio.h>
#include <math.h>
//*****

//*****
int main(void) {
//*****
float a,b,c,x1=1.0,x2=1.0,d,y1,y2;
```

```

int i;
for (i=0; i<20; i++, x2*=10.0F) {
    a=1.0F;
    b=-(x1+x2);
    c=x1*x2;
    d=b*b-4*a*c;
    y1=(-b+(float)sqrt(d))/(2*a);
    y2=(-b-(float)sqrt(d))/(2*a);
    printf("n=%d x1=%g\nx2=%g\ny1="
           "%g\ny2=%g\n\n", i, x1, x2, y1, y2);
}
return 0;
}

```

Ez a program az $(x-1)(x-10^n)=0$ egyenlet gyökeit számolja ki. Előbb $ax^2+bx+c=0$, azaz $x^2-(1+10^n)+10^n=0$ egyenletté alakítja, majd ennek gyökeit veszi az $y_{1/2} = (-b \pm \sqrt{b^2 - 4ac}) / 2a$ képlet segítségével. Ha jól számolna, akkor a gyökök 1 és 10^n volnának, de nem ezt tapasztaljuk. $n=4$ -nél már láthatóvá válik egy kis hiba, 1 helyett 1.00003 a gyök, és a hiba folyamatosan nő: $n=8$ -nál már 2,00294-et kapunk 1 helyett, és az eltérés gyors ütemben nő. A 10^n gyök mindig jó. Ha a float-ot double-re cseréljük, javul a pontosság, de a hiba továbbra is fennáll: $n=16$ -ig jó, de $n=17$ -nél 1 helyett hirtelen 0-t kapunk, és az eredmény ennyi is marad.

A hiba oka a $-b-\text{sqrt}(d)$ kivonás. Itt ugyanis két nagy számot vonunk ki egymásból, melyek között kicsi a különbség.

10.2. C alaptípusok

void ► Azt jelenti, hogy hiányzik. Két esetben használjuk:

- Függvény visszatérési értéke, illetve ha nincs paramétere. Pl. `int main(void){}`, `void kiir(int x)`;
- Típus nélküli pointer (lásd 16. fejezet). Pl.: `void * p = NULL`;

Egész típusok ► A printf/scanf formátumsztringben:

`%[módosító][típusjelző]`

a következő típusjelzőket használhatjuk:

- o: 8-as (oktális) számrendszerben írja ki
- x,X: 16-os (hexadeximális) számrendszerben írja ki. Kis x esetén kisbetűket ír ki, nagy X esetén nagyokat, pl. `0x00face12` ill. `0x00FACE12`.
- d,i: 10-es (decimális) számrendszerben írja ki előjeles egészként.
- u: 10-es (decimális) számrendszerben írja ki előjel nélküli egészként.

Nincs típusjelző a 2-es számrendszerben történő kiírásra.

A következő táblázatban mindenütt a d típusjelző szerepel, de bármelyik behelyettesíthető az előzőek közül.

nem definiált előjel	előjeles	előjel nélküli	méret	scanf és printf
char	signed char	unsigned char	≥ 8 bit	%c mint karakter, %hhc ill. %hhu mint szám.
short signed short short int signed short int	signed short	unsigned short unsigned short int	≥ 16 bit	%hd, %hu
int signed int	signed int	unsigned unsigned int	≥ 16 bit	%d, %u
long signed long long int signed long int	signed long	unsigned long unsigned long int	≥ 32 bit	%ld (el-dé), lu

	long long signed long long long long int signed long long int	unsigned long long unsigned long long int	≥ 64 bit	%lld (el-el-dé), %llu
--	---	---	----------	-----------------------

A C szabvány szerint az adattípusok mérete között a következő reláció áll fenn: $\text{méret}(\text{short}) \leq \text{méret}(\text{int}) \leq \text{méret}(\text{long}) \leq \text{méret}(\text{long long})$. Ennél többet nem tételvezhetünk fel! Az egyes típusok méretére vonatkozóan csak a táblázatban megadott minimális méreteket írja elő a szabvány. Az egy rubrikában lévő elnevezések ugyanazt jelentik. **Az előjeles és előjel nélküli számok mindig egyforma bitszámúak.**

Az egész típusok konkrét jellemzőit a limits.h-ban definiált konstansok segítségével kérdezhetjük le.

Lebegőpontos típusok ► A %e kitevős alakban, a %f tizedes tört alakban, a %g a leginkább olvasható alakban írja ki, beolvasásnál azonos jelentésűek. **A double típusnál eltérő a scanf és a printf formátumsztring!**

típus	pontosság	scanf	printf
float	≥ 6 decimális jegy, ≥ [1e-37, 1e+37], $\epsilon \leq 1e-5$	%e, %f, %g	%e, %f, %g
double	≥ 10 decimális jegy, ≥ [1e-37, 1e+37], $\epsilon \leq 1e-9$	%le, %lf, %lg	%e, %f, %g
long double	≥ 10 decimális jegy, ≥ [1e-37, 1e+37], $\epsilon \leq 1e-9$	%Le, %Lf, %Lg	%Le, %Lf, %Lg

$\text{Méret}(\text{float}) \leq \text{méret}(\text{double}) \leq \text{méret}(\text{long double})$. A lebegőpontos típusok konkrét jellemzőit a float.h-ban definiált konstansok segítségével kérdezhetjük le.

ϵ azt jelenti, hogy 1 és a következő ábrázolható lebegőpontos érték között legfeljebb ekkora a különbség.

Lebegőpontos konstansok:

- float: -1.2f, 3.14F, 0.0f
- double: -1.2, 3.14, 0.0
- long double: -1.2L, 3.14L, 0.0L

Típuskonverzió ► Kisebb pontosságú típusokról nagyobb pontosságúakra gond nélkül konvertálhatunk mind egész, mind lebegőpontos típusoknál, például `int in; long lo; lo=in`; probléma nélkül működik. Bármely lebegőpontos típus nagyobb pontosságúnak számít, mint bármely egész típus.

Ha nagyobb pontosságúról szeretnénk kisebb pontosságúra konvertálni, explicit típuskonverziót kell használnunk, különben a fordító warningot küld. Pl. `in=(int)lo`. Ugyancsak explicit típuskonverzió szükséges, ha előjelesről előjel nélküli vagy fordítva szeretnénk konvertálni.

Példák:

```

int egesz=1;
short rovid=2;
long hosszu=3;
double lebego=4.0;

```

// probléma nélkül másolható:

```

egesz=rovid;
hosszu=egesz;
lebego=egesz;
lebego=10; // Egészből lebegőpontos érték lesz!

```

// adatvesztés lehet:

```

rovid=egesz;
rovid=hosszu;
egesz=10.3;
hosszu=lebego;

```

// adatvesztés lehet, de nem akarunk warningot:

```

rovid=(short)egesz;
rovid=(short)hosszu;

```

```

egesz=(int)10.3;
hosszu=(long)lebego;

// ahol pl. fontos lehet:

egesz=2;
rovid=3;
lebego=egesz/rovid; // lebego értéke 0!
lebego=(double)egesz/rovid; // lebego értéke 0,6666666...

```

Emlékeztetőül: ahol kerek zárójelben megadtuk azt a típust, amivé át akarjuk alakítani az adatot, azt **explicit (közvetlen) típuskonverzió**nak nevezzük. Ahol nem jelöljük, hogy más típusra alakítjuk át az adatot, az az **implicit (közvetett) típuskonverzió**.

Gondolkozzunk együtt!

Oldja meg egyedül!

11. Bitműveletek

Egész számok bitjeivel különböző műveleteket végezhetünk, ehhez számos operátor áll rendelkezésre. A műveleteket két csoportba oszthatjuk: logikai és biteltolás műveletekre. A számok kettes számrendszerbeli alakjával dolgozunk.

A bitenkénti logikai operátorok ► igazságtáblázatai:

A	B	A&B	A	B	A B	A	B	A^B	A	~A
0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	1	1	0	1	1	0
0	1	0	0	1	1	0	1	1	1	1
1	1	1	1	1	1	1	1	0		

ÉS VAGY KIZÁRÓ VAGY NEGÁLÁS

A bitenkénti operátorok a számok azonos helyiértékű bitjei között működnek, a számok minden egyes bitjére.

- Pl.: 11&7 → 001011 & 000111 = 000011 → 3, azaz 11&7==3.
- Pl.: 11|7 → 001011 | 000111 = 001111 → 15, azaz 11|7==15.
- Pl.: 11^7 → 001011 ^ 000111 = 001100 → 12, azaz 11^7==12.
- Pl.: ~7 → ~ 000111 = 111000 → -8, azaz ~7==-8. (Előjel nélküli egésznél a típus bitszámától függ a végeredmény, mert a szám elején minden 0-ból 1 lesz.)

Bitenkénti negálás segítségével egy egész szám mínusz megszerzése **kettes komplement ábrázolásban** így kapható: $-x == (\sim x + 1)$.

A következő azonosság mindig igaz: $(A|B) == (A&B) + (A^B)$.

A bitenkénti operátorok precedenciája kisebb az összehasonlító operátoroknál, emiatt az $\text{if}(x \& y == 3)$ feltétel jelentése: $\text{if}(x \& (y == 3))$, nem pedig $\text{if}((x \& y) == 3)$!!



A rövidebb kód írásának érdekében léteznek az értékadást bitművelettel kombináló operátorok: $|=$, $\&=$, $\wedge=$. Ugyanúgy értelmezettek, ahogy az aritmetikai műveleteknél láttuk: $x|=y \rightarrow x=x|y$, $x\&=y \rightarrow x=x\&y$, $x\wedge=y \rightarrow x=x\wedge y$.

Biteltolás (shiftelés) operátorok ►

A \gg balra tolja a számot, egy bittel való eltolás 2-vel való osztásnak felel meg. Pl.: $19 \gg 2 == 4$. Ha a szám előjel nélküli, vagy előjeles, de nem negatív, akkor 0 bitek lépnek be balról, a jobb oldalon kieső bitek elvesznek. Ha a szám negatív, 1-es bitek lépnek be balról.

A \ll jobbra tolja a számot, egy bittel való eltolás 2-vel való szorzásnak felel meg. Pl.: $19 \ll 2 == 76$. 0 bitek lépnek be jobbról, a bal oldalon kieső bitek elvesznek.

Biteltolásra is léteznek összevont értékadó operátorok, a $\gg==$ és a $\ll==$: $x \gg== y \rightarrow x = x \gg y$, $x \ll== y \rightarrow x = x \ll y$.

A biteltolás során az egyik oldalon kieső bitek nem jönnek vissza a másik oldalra.

$x \ll y$ művelet nem változtatja meg x értékét. Ha változást akarunk, pl. így kell: $x \ll== y$.



```

Mit ír ki?
int a=3,b=6;
printf("%d\t%d\t%d\t%d\n", a&b, a|b, a^b, ~a);

```

Hány bites egy egész? ► Bitműveletek segítségével határozzuk meg, hogy hány bites egy egész szám azon a számítógépen, ahol a programot futtatják!

1-et teszünk egy változóba, és addig toljuk balra, amíg végül kiesik a túloldalon.

Bitszámláló program:

Tegyéél 1-at a változóba!
A számláló legyen nulla!
Amíg a változó nem nulla
Told balra egy bittel a változót!
Növeled meg a számlálót eggyel
VÉGE.

```
#include <stdio.h>

int main(void) {
    int változo=1, szamlalo=0;
    while (változo!=0) {
        változo<<=1;
        szamlalo++;
    }
    printf("%d\n", szamlalo);
    return 0;
}
```

Írjuk át úgy a programot, hogy biteltolás helyett szorzást használ!

Pakolt egész példa ► Egy feladat során nagyon sok 0 és 15 közötti egész számot kell tárolnunk. A 0 és 15 közötti számok elférnek 4 biten, azonban a C nyelvben található legkisebb adattípus, a char is egy bájtos, azaz legalább 8 bites. Nem szeretnénk pazarolni a memóriát, ezért olyan függvényeket írunk, amelyek két darab ilyen számot tesznek bele egy unsigned char típusú változóba, illetve kiveszik onnan. A függvények bitműveletekkel dolgoznak.

A betevő függvény jellemzői:

- Három paramétere van:
 - az egész érték, amit be kell tennie,
 - hová kell tennie (0: alulra, 1: felülre),
 - a módosítandó unsigned char érték
- Először a bemenő értéket maszkolja 4 bitre.
- Létrehozza a pakolt értéket biteltolás és maszkolás segítségével.
 - Ha alulra kell tennie: (módosítandó&11110000)|bemenő
 - Ha felülre kell tennie: (módosítandó&00001111)|(bemenő<<4 bit)

Például felülre a 6-ot:

```
10101010=170 a módosítandó
& 00001111=15 (azaz 0xf) a maszk
-----
00001010=10 (felül 0, alul maradt, ami volt)
| 01100000=(00000110=6 bemenő)<<4
-----
01101010=106, a pakolt érték
```

A kivevő függvény jellemzői:

- Két paramétere van:
 - a pakolt érték
 - honnan kell kivennie (0: alulról, 1: felülről),
- Maszkolás és biteltolás segítségével kiveszi a számot.
 - Ha alulról kell kivennie: pakolt&00001111
 - Ha felülről kell kivennie: (pakolt&11110000)>>4 bit

Például a felsőt kivesszük:

```
01101010=106, a pakolt érték
&11110000=240 (azaz 0xf0) a maszk
-----
01100000=96 → (01100000>>4)=00000110=6
```

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

//*****
unsigned char bepakol(int be, int hova,
                     unsigned char mod){
//*****
    be &= 0x0f; //így biztos csak az alsó 4 bit lehet nem 0
    switch(hova){
        case 0: return (mod & 0xf0) | be;
        case 1: return (mod & 0x0f) | (be << 4);
    }
    return 0;
}

//*****
int kikapol(unsigned char pakolt, int honnan){
//*****
    switch(honnan){
        case 0: return pakolt & 0x0f;
        case 1: return (pakolt & 0xf0) >> 4;
    }
    return 0;
}

//*****
int main(void){
//*****
    int be1=11, be2=6;
    unsigned char tarolo;
    tarolo=bepakol(be1, 0, 0);
    tarolo=bepakol(be2, 1, tarolo);
    printf("Tarolo=%llu\n", tarolo);
    printf("ki1=%d\n", kikapol(tarolo, 0));
    printf("ki2=%d\n", kikapol(tarolo, 1));
    return 0;
}
```

Gondolkozzunk együtt!

Hány 1-es bit van az egész számban?

Oldja meg egyedül!

12. Bemenet és kimenet, ASCII

A fejezetben először a karakterek kódolására használt ASCII szabvánnyal ismerkedünk meg, ezt követően megnézzük néhány szabványos függvényt, melyekkel karaktereket illetve szövegeket tudunk beolvasni illetve kiírni. Végül a printf és a scanf függvényeket tekintjük át részletesebben.

12.1 A karakterek ASCII kódja

A számítógép memóriája bitekből, azaz kettes számrendszerbeli számokból áll. Ahhoz tehát, hogy szövegekkel lehessen dolgozni, a szövegeket valamiképpen számokkal kell tárolni. A szövegekben előforduló minden karakterhez hozzárendeltek egy számot. Kezdetben minden cég a saját szám-karakter párosaival dolgozott, később a szabványosított ASCII (American Standard Code for Information Interchange, ejtsd: [æski], magyarosabban „észki”) kiszorította a többi variációt, és napjainkig meghatározza a karakterkódolást. Csak az ASCII-re épülő UNICODE illetve ennek változatai (UTF-8, stb.) voltak képesek részben kiszorítani, azonban a C nyelvű programok még ma is zömmel ASCII-ben íródnak.

Az ASCII szabvány 7 biten ábrázolja a karaktereket, azaz a 0-127 számokhoz rendel karaktert. A számtartomány két részre oszlik: 0 és 31 között található a vezérlő karakterek (ezeket leginkább nyomtatók vezérlésére használták), mint az újsor, koci vissza, vízszintes és függőleges tabulátor, lapdobás, escape, stb. A 32-126 tartományban vannak a látható karakterek, melyeket az alábbi táblázatban foglaltunk össze. A 127-es karakter a delete.

32-47:	! " # \$ % & ' () * + , - . /
48-63:	0 1 2 3 4 5 6 7 8 9 : ; < = > ?
64-79:	@ A B C D E F G H I J K L M N O
80-95:	P Q R S T U V W X Y Z [\] ^ _
96-111:	` a b c d e f g h i j k l m n o
112-127:	p q r s t u v w x y z { } ~ Δ

A táblázatban csak az angol ABC betűit találjuk, hiszen ez egy amerikai szabvány. Mi a helyzet a magyar karakterekkel? A C programunkban tudunk ékezetes betűket írni. Ez úgy lehetséges, hogy bár az ASCII 7 bites rendszerekre készült, később a legalább 8 bites bájtokat alkalmazó számítógépek terjedtek el, a nyolcadik bit pedig duplájára növeli a használható karakterek számát. A 128-255 közötti számtartomány azonban nem szabványos, sokféle kódtábla van használatban. Ennek következménye az a furcsa jelenség, hogy ha Visual C++-ban írunk egy programot, és abban magyar karaktereket tartalmazó szöveget helyezünk el, akkor a konzol ablakban hibásan jelenik meg a kiírt szöveg. (printf("Helló világ!\n"); → Hell`világ!).

Egy másik következmény, hogy ha neveket rendezünk ABC sorrendbe, akkor a rendezést a karakterekhez rendelt számok nagysága alapján végezzük, emiatt a magyar magánhangzók az összes angol betű után kerülnek. Ezzel a jelenséggel általában nem szoktunk foglalkozni. Ha valamiért kifejezetten fontos lenne a magyar magánhangzók helyes kezelése, saját algoritmust kell készíteni. Ezt az olvasóra bízunk (mire eljutunk a rendezésekig, már minden szükséges tudással rendelkezni fog).

A Visual C++-ban programozók számára hasznos lehet az alábbi függvény, amely a paraméterként kapott sztringben lecseréli az ékezetes magyar betűket a konzol ablakban helyes változatokra, így a példában látható „Helló világ!” helyesen jelenik meg. A függvény működését nem magyarázzuk, mert még nem mindent tudunk hozzá. Javasoljuk, hogy ha az olvasó eljutott a 19. fejezet végéig, térjen vissza ide, és értse meg a működést! Addig is használja nyugodtan!

```

/*****
#include <stdio.h>
*****/

/*****
void magyarit(char * src){
*****/
    unsigned i;
    for(i=0;src[i];i++){
        switch(src[i]){
            case 'á': src[i]=160; break;
            case 'é': src[i]=130; break;
            case 'í': src[i]=161; break;
            case 'ó': src[i]=162; break;
            case 'ö': src[i]=148; break;
            case 'ő': src[i]=139; break;
            case 'ű': src[i]=163; break;
            case 'ü': src[i]=129; break;
            case 'ú': src[i]=251; break;
            case 'Á': src[i]=181; break;
            case 'É': src[i]=144; break;
            case 'Í': src[i]=214; break;
            case 'Ó': src[i]=224; break;
            case 'Ö': src[i]=153; break;
            case 'Ő': src[i]=138; break;
            case 'Ű': src[i]=233; break;
            case 'Ü': src[i]=154; break;
            case 'Ú': src[i]=235; break;
        }
    }
}

/*****
int main(void){
*****/
    char szoveg[100]="Helló világ\n";
    magyarit(szoveg);
    printf("%s",szoveg);
    return 0;
}

```

Figyeljük meg, hogy a számjegyek mint karakterek a 48-57 sávban találhatóak, azaz '0'!=0. A sztringek végét jelző '\0' nem egyezik meg az '0'-val, megegyezik viszont a 0 számmal, azaz '\0'==0!



Mivel a 0-31 ASCII karakterek vezérlőkarakterek, ezeket nem lehet egyszerűen beírni sztringekbe, de az idézőjel vagy aposztróf is gondot jelent. Ezek elhelyezésére a \ karakter után írt azonosító segítségével lehetséges. A következő táblázat a C-ben használható speciális karakterkonstansokat tartalmazza:

Megadás	Jelentés
\a	Sípoló hang
\b	Backspace
\f	Lapdobás (form feed)
\n	Új sor
\r	Kocsi vissza (visszamegy a sor elejére, és az új szöveg felülírja a régit)
\t	Tabulátor
\v	Függőleges tabulátor
\'	Aposztróf
\"	Idézőjel
\\	Backslash
\?	Kérdőjel (a \ nélkül írt ? is jó)
\ooo	ASCII karakter nyolcas (oktális) számrendszerben megadva
\xhh	ASCII karakter tizenhatos (hexadecimális) számrendszerben megadva

A fenti speciális karakterek minden sztringben használhatók. A printf és a scanf formátumsztringjében a % jel különleges

karakter, ezt követően adjuk meg a felsorolt változó típusát (printf-nél a %% írja ki magát a % jelet). A % jel más esetben nem speciális karakter, azaz pl. a `printf("%s", "x=%d%\n");` az `x=%d%%` szöveget írja ki!

Példa ▶ Írjuk ki a karakterek ASCII kódját!

```
#include <stdio.h>

int main(void) {
    int i;
    for(i=0; i<255; i++)
        printf("%c=%d\t", i, i);
    return 0;
}
```

A programot futtatva azt tapasztaljuk, hogy a vezérlőkarak-
terek egy részét nem látjuk, a 9 értéket pedig a 13 követi. A 9 az
újsor karakter, a '\n', a 13 pedig a kocszi vissza, azaz a '\r'. A '\r'
hatására a kiírás visszaugrik a sor elejére, és felülírja a korábban
kiírt szöveget, ezért nem látszanak a 10-12 karakterek. (Nagy
számításgényű programoknál a '\r' remekül használható az előre-
haladás kiírására: 1%, 2%...)

Figyeljük meg, hogy int típusú változót jelenítünk meg karak-
terként! **A karaktereket tárolhatjuk int-ben is**, kivéve, ha
sztringről van szó. A sztringeket mindig char tömbben tároljuk.

12.2 getchar/putchar, gets/puts, fgets

A szabványos bemenetről való beolvasásra illetve írásra álta-
lában a scanf/printf párost használjuk, mert ezekkel szinte minden
feladatot meg tudunk oldani, van azonban néhány speciális függ-
vény a C-ben, melyek szintén a felhasználóval történő kommuni-
kációra szolgálnak.

getchar/putchar ▶ Egy karaktert olvashatunk be a
`scanf("%c", &c);` helyett a `c=getchar();` utasítással (c
char típusú). A két változat teljesen egyenértékű, bármelyiket
használhatjuk.

A hibakezelés másképpen történik a két függvény esetében. A
scanf a sikeres beolvasások számával, vagy az EOF konstanssal
tér vissza. A scanf esetén így ellenőrizhetjük a beolvasás sikerét:

```
if (scanf("%c", &c) != 1) {
    fprintf(stderr, "Hibas beolvasas!\n");
    exit(1);
}
```

A scanf tehát 1-et ad vissza, ha sikerült beolvasni a karaktert.
(Az fprintf függvény fájlba írásra szolgál, jelen esetben azonban
nem valódi fájlba dolgozik, hanem egy fájlnak látszó dologba, a
szabványos hiba kimenetre, vagyis az stderr-re, ami normál
esetben a képernyő.) Az `exit(1)` kilép a programból, azaz vége a
programunknak.

A `getchar` esetében nincs külön paraméterként kapott célvál-
tozó és külön visszatérési érték, mint a scanf esetén, ezért a hibát
a visszatérési értéknek kell tartalmaznia. A C nyelv készítői a
következő trükköt alkalmazzák: a `getchar` nem char típusú értéket
ad vissza, hanem `int`-et. Ha nem történt hiba, akkor ez az érték
beírható egy char típusú változóba, ha azonban hiba történt, a
`getchar` az EOF konstanszt adja vissza, melynek értéke általában -
1 szokott lenni. Az EOF nem char típusú érték, tehát char típusú
változóba nem írható (a `char c=EOF;` hatására a c-be bekerül
egy érték, ami már nem az EOF, mert ha ezt követően egy `int`
`i=c;` `if (c==EOF)...` utasítássorozatot adunk ki, fordítótól
függ, hogy az if feltétele IGAZ-nak vagy HAMIS-nak bizonyul-e.

A `getchar`-t így használhatjuk hibellenőrzésre:

```
int ch=getchar();
if (ch==EOF) {
```

```
fprintf(stderr, "Hibas beolvasas!\n");
exit(1);
}
```

A `getchar` párja a `putchar`, mely a `printf("%c", c);` -nek
felel meg: `putchar('X');` `putchar('\n');` `put-
char(ch);`

Általában kiírásnál, így a `putchar`-nál sem szoktunk hibát el-
lenőrizni. Beolvasás esetén azért fontos, mert a felhasználó EOF-
fal is jelezheti, hogy nem kíván több adatot megadni (ha Ctrl+Z-t
nyomunk a billentyűzeten, amikor valamelyik beolvasó függvény
adatokra vár éppen, akkor ezzel EOF-ot küldünk a beolvasó
függvénynek). Az EOF egyébként az *End Of File*, azaz *fájl vége*
kifejezés rövidítése, és fájlból olvasásnál a fájlolvasó függvények
is ezt adják vissza. A billentyűzetről való olvasás sokban hasonlít
a fájlból való olvasására, ezért használják itt is ezt a konstanszt. Az
EOF az `stdio.h`-ban van definiálva, használatához szükséges az
`stdio.h` `#include`-dal való beszerkesztése.

A `scanf` függvény esetén pedig további szerepe a hibakeze-
lésnek, hogy amennyiben számot próbálunk beolvasni, és a fel-
használó nem számot ad meg, a `scanf` függvény hibajelzést ad
vissza. Ha ezzel nem törődünk, akkor a programunk hibásan fog
működni. A `scanf` függvény esetén például így kezelhetjük a
hibát:

```
int i;
printf("Adj meg egy egész számot! ");
while (scanf("%d", &i) != 1) {
    fprintf(stderr, "Nem szám.\n");
    scanf("%*s");
    printf("Adj meg egy egész számot! ");
}
```

A kódrészlet érdekessége a ciklusban megbújó
`scanf("%*s");`. A `%*s` azt jelenti, hogy egy sztringet olva-
sunk be, de azt nem tároljuk el sehol. A `*`-ot a többi adattípusnál
is használhatjuk, hasonló jelentéssel: beolvassa, de nem tárolja.
(Megjegyzés: a fenti kód az EOF-ot nem kezeli.)

gets/puts ▶ A `gets` függvény sztring beolvasására szolgál. A
`scanf("%s", s);` és a `gets(s);` között jelentős különbség
van: a `gets` teljes sort olvas be, míg a `scanf` csak a következő
whitespace karakterig olvas, ezért a `scanf`-et ezen a módon nem
tudjuk név beolvasására használni. (A 12.3 alfejezetben látjuk
majd, hogy ez megoldható a `scanf`-fel is.)

A `gets` használata:

```
char s[100];
gets(s);
printf("%s", s);
```

A `printf` helyett használhatjuk a `puts` függvényt is:

```
puts(s);
```

A két kiírás között az az eltérés, hogy a `puts` tesz egy újsort a
kiírt szöveg után. (Ha `printf("%s\n", s);` -t használunk,
ugyanaz az eredmény.)

Figyeljük meg, hogy a `getchar` visszatérési ér-
téke a beolvasott karakter, míg a `gets` a paramé-
terként kapott karaktertömbbe teszi az eredményt!



A `gets` visszatérési értéke `NULL` konstans, ha nem sikerült a
beolvasás:

```
if (gets(s) == NULL) {
    fprintf(stderr, "Hibas beolvasas!\n");
    exit(1);
}
```

fgets ▶ A `gets` esetén probléma, hogy ha a felhasználó hosszabb
szöveget ad meg, mint a karaktertömb mérete, akkor a `gets` a
tömb utáni memóriarészre próbálja meg betölteni a sztring végét,
ahol valószínűleg más változók vannak, melyeket felülír. A másik
lehetőség, hogy a programunk hibáüzenettel elszáll. Ugyanez a
probléma fennáll a `scanf` esetében is, azonban a 12.3 alfejezetben

látjuk majd, hogy *scanf*-nél korlátozható a beolvasható karakterek száma.

Az *fgets* függvényt eredetileg szöveges fájlból való beolvasásra találták ki, azonban használhatjuk a szabványos bemenet olvasására is. Ez a függvény szintén egy sornyi szöveget olvas, mint a *gets*, azonban van egy paramétere, mellyel megadhatjuk a karaktertömb méretét, ahová beolvassa. Ha a felhasználó túl hosszú szöveget ad meg, az *fgets* nem olvassa be a szöveg végét, hanem otthagyja egy következő beolvasás számára. (A megadott méret tehát a karaktertömb mérete, az *fgets* ennél egy karakterrel kevesebbet enged meg, mert a lezáró nullának is hagy helyet. A *scanf*-ben megadható korlát viszont a karakterek számát jelenti, a *scanf* nem figyel a lezáró nullára!)



Példa az *fgets*-sel történő beolvasásra:

```
char s[100];
fgets(s, 100, stdin);
puts(s);
```

12.3 printf és scanf

Belső formátum: ahogy a számítógép tárolja az adatokat

Külső formátum: ahogy az adatok az ember számára érthetőek.

A *printf* és *scanf* függvények a külső és belső formátum közötti átalakítást végzik.

Pl. a 123 az ember számára 3 számjegy, és a képernyőn 3 karakter ábrázolja. A számítógép belső formátumában a 123 egy darab egész számként, kettes számrendszerben tárolódik. Ha például a *scanf* beolvassa egy *int* típusú változóba az általunk megadott 123-at, akkor ő három darab karaktert kap ('1', '2', '3'), ebből előállítja az 1, 2 és 3 értéket ('1'-0, '2'-0, '3'-0), majd kiszámítja, hogy $1 \cdot 100 + 2 \cdot 10 + 3$ a szám értéke, és ezt raktározza el. Emlékezzünk: az '1' ASCII kódja 49, a '0' ASCII kódja 48, a kettő különbsége valóban 1.

Ez a két függvény rokonfüggvényeikkel együtt (*fprintf*, *sprintf*, stb.) a C legbonyolultabb függvényei. Paraméterlistájuk is olyan, amelyet nagyon ritkán használunk: változó paraméterlistával rendelkeznek, azaz paramétereik száma és típusa nem rögzített (lásd a 33. fejezetet). Egyszerűsített prototípusuk a következő:

- `int printf(char formatum[],...);`
- `int scanf(char formatum[],...);`

Az első paraméter mindkét esetben a formátumsztring. A formátumsztring olyan sztring, mely leírja a kiírandó szöveg, és a benne lévő paraméterek formátumát, illetve a beolvasandó paraméterek formátumát.

A második paraméter mindkét esetben A három pont azt jelzi, hogy ide tetszőleges számú, tetszőleges típusú paramétert megadhatunk. A tetszőleges szám 0 is lehet.

A *scanf* függvény csak beolvas, nem ír ki semmit! Ha ezt íránk: *scanf("Add meg a számot! %d", &a);*, ez hibásan működne, mert a *scanf* nem tudja értelmezni az „Add meg a számot”, részt, és ezt kiírni biztosan nem fogja (illeszteni próbál rá, lásd később).

Mindkét függvény visszatérési értéke *int*. A *printf* visszaadja, hogy hány karaktert írt ki. A *scanf* vagy azt adja vissza, hogy hány darab változót sikerült beolvasni, vagy egy speciális érték, amit az EOF nevű konstans definiál, mely szintén a beolvasás sikertelenségét mutatja. A *printf* visszatérési értékét általában nem használjuk.

Figyelem! Sem a *printf*, sem a *scanf* nem tudja ellenőrizni, hogy a programozó hány darab és milyen típusú változót sorolt fel a formátumsztring után. Mindig feltételezik, hogy ott azt kapják, amit várnak. Ha hibásan adjuk meg, a program akkor is lefordul, de rosszul fog működni. Ez olyan hibát is eredményezhet, ami csak időnként jelentkezik, vagy más platformon (gép/operációs rendszer) lefordítva okoz hibát.



A szakasz további részében leírtakat nem kell tudni, ha szükség van rá, elég fellapozni ezt a részt, vagy megnézni a **Help**-ben.

Nézzük meg, hogyan használhatjuk a %valami tagokat a két függvényben!

printf: % [flagek][méret][pontosság][típushossz]típus

A [] közötti részek nem kötelezőek.

Részletek:

- típus:
 - o d, i: előjeles egész, a d és az i ugyanazt jelenti, azaz bárhol írhatjuk a %d helyett a %i-t
 - o u: előjel nélküli (azaz nemnegatív) egész
 - o o: nemnegatív érték kiírása 8-as (oktális) számrendszerben
 - o x, X: nemnegatív érték kiírása 16-os (hexadecimális) számrendszerben. Kis x esetén kisbetűt, nagy X esetén nagy betűt ír ki, pl. %x ill. %X esetén 14-et e-nek ill. E-nek írja ki.
 - o p: pointer kiírása hexadecimális formában
 - o e, E: valós szám kiírása kitevős (exponenciális) alakban, pl. %e és 1000.0 eredménye: 1.000000e+003, azaz 1×10^3 ; %E esetén 1.000000E+003 az eredmény.
 - o f: valós szám kiírása fixpontos alakban, azaz 1000.0 eredménye 1000.000000.
 - o g, G: valós szám kiírása általános (general) formában: az ember számára legjobban olvasható formát választja. 1000.0 eredménye 1000.
 - o c: karakter
 - o s: sztring
 - o n: ez egy érdekes típus, mert nem ír ki semmit, hanem egy *int* * pointer-t vár, és a mutatót *int*-be beírja, hogy eddig a %n-ig hány karaktert írt ki. Pl.:

```
printf("Mit sutsz kis szucs?\n"
      "Sos hust sutsz, kis szucs? ", &a);
printf("%d\n", a);
```

eredménye „Mit sutsz kis szucs? Sos hust sutsz, kis szucs? 20”.
 - típushossz:
 - o h: egészeknél a short típusokhoz, pl. %hd, %hx
 - o hh: egészeknél a char típusok számként kezeléséhez: %hh
 - o l (kis L): egészeknél a long típusokhoz, pl. %ld, %lx
 - o ll: egészeknél a long long típushoz (régbebi C fordítók nem ismerik), pl. %lld, %llx
 - o L: valósaknál a long double típushoz, pl. %Lf (a float és double típusokat egyaránt %e, %f ill. %g módon írjuk ki)
 - méret (a kiírás minimális szélessége):
 - o egész szám: ennyi karakternyi helyre írja ki az értéket (számot, betűt vagy szöveget), a tartományon belül jobbra igazítva. Az üres részt alapesetben szóközzel tölti fel. (Ezzel a módszerrel érhető el az állandó oszlopszélességű kiírás.) Ha a kiírandó érték nagyobb, mint a megadott hely, akkor nem vágja le, hanem nagyobb lesz a kiírás: akkora, amekkora szükséges. Pl. %3d és 12 eredménye „ 12”, vagyis 1 db szóköz van előtte.
 - o *: paraméterként megadható a kiírandó szélesség. Pl.: `printf("%*d", 3, 12);` eredménye: „ 12”, vagyis 1 db szóköz van előtte.
 - .pontosság (más jelentés az egyes típusoknál)
 - o d, i, u, o, x, X: a kiírt számjegyek minimális száma (0-kat ír a szám elé, ha az kisebb)
 - o e, E, f: a tizedespont utáni számjegyek száma, pl. `printf("%10.2e\n", 12.789);` eredménye: „ 1.28e+001”, egy szóköz van az elején, két tizedes jegyre kerekít.
 - o g, G: az értékes jegyek száma (ha az egész rész hosszabb, nem csonkol, pl. `printf("%2g\n", 123456.789);` eredménye 123457).
 - o s: a kiírt karakterek maximális száma (csonkol). Pl. `printf("%.4s\n", "123456.789");` eredménye: 1234
 - flagek (több is használható egyszerre)
 - o -: balra igazítja a kiírt értéket. Pl. %-4dx és 12 eredménye „12 x”, az x előtt 2 szóköz.
 - o +: mindenképp ír ki előjelet, pozitív számok esetén is. Pl. %+d és 12 eredménye +12
 - o 0: szóközők helyett 0-kal tölti fel a kiírt szám előtti részt, ha a szélesség nagyobb, mint a kiírandó érték. Pl. %03d és 12 eredménye: 012. A - flag esetén figyelmen kívül hagyja a függvény. Pl. %-03d eredménye: „12”.
 - o #: oktális számnál 0-t, hexánál 0x-et vagy 0X-et ír a szám elé, valósaknál mindenképp kiírja a tizedes pontot.
- scanf: %[*][méret][típushossz]típus**
- típus (sok hasonlóság, de sok lényeges eltérés van a *printf*-hez képest!)
 - o d: előjeles egész szám 10-es (decimális) számrendszerben
 - o i: előjeles egész szám, 8-as, 10-es vagy 16-os számrendszerben (12: decimális, 012: oktális, 0x12: hexa)
 - o u: nemnegatív (előjel nélküli) decimális egész
 - o x: nemnegatív hexadecimális egész
 - o o: nemnegatív oktális egész
 - o c: karakter
 - o e, E, f, g, G: lebegőpontos szám, float típusú. Az ötből bármelyiket használhatjuk, a felhasználó bármilyen módon beírhatja a számot, akár kitevős alakban, akár fixpontos alakban. **Fontos:** a float számok beolvasása %e, %f, %g stb, a double számok beolvasása %le, %lf, %lg! Ha nem így adjuk meg, a program nem fog jól működni! (*printf*



esetén a float és a double típusnál egyaránt a %e, %f, %g alakot használjuk).

- o s: sztring
- o p: pointer
- o n: ugyanaz, mint a printf-nél, a beolvasott karakterek számát tárolja.
- o []: lásd lejjebb, az illesztéseknél.
- típushossz
 - o h: short egészekhez, pl. %hd
 - o l: long egészekhez, pl. %ld
 - o l: double valóshoz: %le, %lE. %lf, %lg, %lG (mind az öt azonos jelentésű, felcserélhető)
 - o ll: long long egészhez (régbebi fordítók nem ismerik)
 - o L: long double valóshoz
- méret: egész szám, a beolvasott és konvertált karakterek maximális számát adja. Pl. a `scanf("%2d %2d", &a, &b);` eredménye, ha 87654-et ad meg a felhasználó: a=87, b=65 lesz. Egy következő `scanf` kapja a 4-et.
- *: dobjuk el a felhasználó által megadott értéket. A *-os beolvasáshoz nem is tartozik változó a paraméterlistán. Pl. `scanf("%*d", &a);` eredménye, ha 12 34-et ad meg a felhasználó: a=34, az 12-t eldobtuk.

Illesztések a scanf használatával ► Illesztés lehetséges a %-os kifejezésen belül, illetve azon kívül.

- A %[] formátum szűrt sztring beolvasására alkalmas
 - o %[abcd]: Addig olvassa be a szöveget, amíg abban olyan karakterek vannak, amik a zárójelben. Pl. ha `scanf("%[badluck]", s);` a beolvasás, és a felhasználó azt adja meg, hogy „addoda!”, akkor a beolvasott szöveg „add”, mert az o nincs a zárójelben.
 - o %[A-Za-z]: Tartomány megadása. A-Z-ig és a-z-ig a betűket, valamint a szóközt olvassa be. Így csak angol betűkből álló emberek nevét tudjuk beolvasni, hiszen a magyar magánhangzók nincsenek benne az ASCII A-Z ill. a-z tartományában.
 - o %[^\A-Z]: Minden, ami nem az A-Z tartományban van. A halmazra nem illeszkedő karaktereket olvassa be. A két változat nem kombinálható, azaz a %[A-Z^\QX] nem azt jelenti, hogy A-Z-ig illeszt, kivéve a Q és X, hanem azt, hogy A-Z-ig, továbbá ^ is lehet benne (meg a Q és az X is, bár ezeket tartalmazza az A-Z tartomány is).
 - o Magyar nevek beolvasása: %[^\n] módon megadva a sor végéig olvas, így elkerülhető a gets használata: `scanf("%[^\n]", nev);`
 - o %[A-Z-]: Az A-Z tartomány, plusz a [] zárójelpár, és a - jel is lehet a szövegben. A] csak az első lehet a nyitó zárójel után, a - pedig csak az utolsó a záró zárójel előtt. A [bárhol lehet.
- A % formátumon kívüli karakterekre pontosan illeszkedő bemenetet vár a scanf. Ha nem azt kapja, akkor sikertelen lesz a beolvasás.
Pl. `scanf("Születési év: %d", &a);`
Ha a felhasználó ezt írja be: „Születési év: 1817”, akkor a-ba bekerül 1817.
 - o A szóköz speciális: bármely whitespace karaktert helyettesítheti. Pl. a "%d %d" esetén az első szám beírása után nyugodtan üthetünk ENTER-t, és úgy adhatjuk meg a második számot. Sőt, akár így is írhatuk volna a formátumsztringet: "%d%d", ez ugyanazt jelenti. Az előző példában pedig a felhasználó nyugodtan írhatja külön sorba a Születési, az év: és az 1817-et, úgy is elfogadja a scanf.
 - o Ha azt akarjuk, hogy a scanf ugorjon át mindent, amit a felhasználó beírt, és az első számot olvassa be, akkor pl. `scanf("%*[0-9]%d", &a);` módon járhatunk el. Ez így beolvassa a negatív számokat is, viszont ha a felhasználó - karaktert írt valahol nem előjelként, akkor sikertelen lesz a beolvasás.
 - o Egy szótár minden sorában angol=magyar módon szerepelnek mondatok. Milyen formátumsztringgel olvassuk be, ha csak a magyar mondatokra van szükségünk?
Megoldás: `scanf("%*[%=\n]", &mondat);`



13. A számítógépek felépítéséről és működéséről

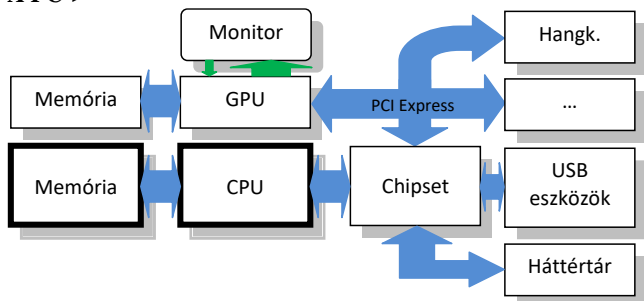
Neumann elvek: A számítógép működését a tárolt program elve határozza meg, azaz a gép a program utasításait és az adatokat egységesen, a központi memóriában, bináris formában tárolja. Az ezen elv szerint működő számítógépek univerzális Turing gépet valósítanak meg és soros utasítás-végrehajtást alkalmaznak.

A fenti azt jelenti, hogy a processzor az adatot is és az utasítást is a memóriából olvassa be, az utasítást elemzi, és az utasításnak megfelelően dolgozik tovább. Korábban az utasításokat kevésbé rugalmas módon tárolták, pl. fixen beállították a számítógép indítása előtt.

Neumann architektúra: Neumann elvek alapján felépített számítógép, az adatokat és utasításokat ugyanazon az adatbuszon olvassa be a processzor vezérlőegysége. A PC és a legtöbb számítógép ezt az architektúrát valósítja meg.

Harvard architektúra: Ugyancsak Neumann elvek szerint épül fel, viszont az utasításokat és adatokat fizikailag elválasztva kezeli, külön buszon fér hozzájuk a CPU. Mikrokontrollerekben igen elterjedt.

A PC ►



A buszok ► kommunikációs csatornák, melyeken információ áramlik a számítógép egyes részei között, de egyes egységeken belül is lehetnek buszok. Számos eltérő típusuk létezik.

- A buszokon általában kettőnél több eszköz található, ezek közül egyszerre általában egy küld, egy pedig fogad információkat. A buszvezérlő áramkörök dolga, hogy kiválasszák, mikor ki kommunikálhat.
- Megkülönböztetünk soros és párhuzamos működésű buszt. Soros busznál a bitek egymást követik ugyanazon a vezetéken, párhuzamos busznál az adatok több vezetéken haladnak egyszerre, párhuzamosan. Régebben a lassú buszok voltak gyors felépítésűek, a gyors buszok pedig párhuzamosak. Manapság a lassú és gyors buszok sorosak, a közepes sebességűek párhuzamosak. Ennek az az oka, hogy nagy sebességnél az egyszerre elküldött bitek nem egyszerre érkeznek meg a címzetthez, már akár néhány centiméter hosszú vezetékek esetében sem. A nagysebességű soros buszok alkalmazását a modern áramköri technika tette lehetővé. Soros buszok esetén is létezik párhuzamosítás: több soros vezetéken küldik az információt, itt azonban bonyolult küldő és fogadó áramkörök szükségesek. PCI Express busz esetén az 1×, 2×, 4×, 8×, 16×, 32× ilyen párhuzamosítással éri el a nagyobb sebességet.
- A CPU és a memória között külön busz viszi az adatokat, a memóriacímeket és a vezérlő információkat (adatbusz, címbusz, vezérlő busz).
- USB = Universal Serial Bus = Általános Soros Busz

Memória és háttértár ► A számítógépek fő funkciója az adatfeldolgozás. Az adatokat és a feldolgozó utasításokat valahol tárolni kell. Általában elmondható, hogy minél több információ tárolására alkalmas egy eszköz, annál lassabb. Továbbá általában gyorsabbak azok az eszközök, melyek illékonyak, azaz elveszítik tartalmukat, ha megszűnik az elektromos tápellátás. A háttértárak nagyméretűek, nem illékonyak, viszont túl lassúak ahhoz, hogy a futó programot itt tároljuk. A memória és a háttértár egyaránt lehet csak olvasható, valamint írható-olvasható.

ROM: Read Only Memory (csak olvasható memória). Minden számítógépben találunk ilyet. A számítógép indulásakor az ebben található programkód indítja az operációs rendszer betöltését. A PC-n ebben található a BIOS (Basic Input-Output System). Mikrokontrollereknél gyakran a működtető program és a konstans adatok is ROM-ban találhatóak, és a RAM mérete nagyon kicsi. ROM-nak szokás nevezni azokat a memóriákat is, melyek nem illékonyak, azonban lehetővé teszik a tartalmuk megváltoztatását speciális körülmények között (EPROM, flash-ROM).

RAM: Random Access Memory (tetszőleges elérési memória). A rövidítés általában vonatkozhatna a ROM-okra vagy a merevlemezre is, hiszen ott sem csak sorban egymás után érhetjük el az adatokat, ahogy egy mágnesszalagos tárolónál, hanem bárholonnan olvashatunk vagy írhatunk. A RAM elnevezést ennek ellenére csak az írható-olvasható memóriára használjuk, mely általában illékony, de nem feltétlenül. Legelterjedtebb a statikus RAM (SRAM) és a dinamikus RAM (DRAM).

A DRAM viszonylag lassú, de a szilíciumból megvalósított cellái sokkal kisebb helyfoglalásúak, mint a SRAM cellái. A DRAM-ban egy kis integrált kondenzátor tárolja a bitet, melyből az elektronok elszivárognak, emiatt a segédáramkörök

Gondolkozzunk együtt!

Oldja meg egyedül!

másodpercenként sokszor kiolvassák és visszairják a memóriatartalmat, azaz frissítik az adatot. Emiatt dinamikus a dinamikus RAM. A DDR memóriáknál olvasható több száz vagy több ezer MHz-es órajel becsapós, mert az egyes DRAM cellák ennél sokkal lassabbak, mindössze néhány MHz-es, esetleg néhány tíz MHz-es írási/olvasási sebességet tesznek lehetővé. A nagyobb sebességet azzal érik el a fejlesztők, hogy a mátrixban tárolt RAM cellák egy teljes sorát, általában több tucat, akár több száz kilobitot töltenek be egyszerre egy átmeneti SRAM pufférbe (nevezhetjük akár cache-nek is), és innen már valóban lehet GHz-es sebességgel írni/olvasni. Ez azt jelenti, hogy amennyiben nem egymás után következő adatokat vagy utasításokat olvasunk, hanem össze-vissza ugrálunk a memóriában, durván lecsökken a programfutás sebessége.

A SRAM általában egy 4 vagy 6 tranzistorból felépült flip-flop áramkör, ugyanolyan gyors lehet, mint a processzor. A DRAM-nál 4-6-szor nagyobb helyfoglalás miatt átmeneti tárként, vagy cache számára használják. Vannak elképzelések egy tranzistoros (1T) RAM-ok kifejlesztésére is. A regiszterek is SRAM jellegű cellákban tárolják a biteket, melyeket segédáramkörökkel egészítenek ki azért, hogy a regiszterekben lévő adatokkal a processzor a lehető leggyorsabban tudjon dolgozni.

CPU ► Central Processing Unit (központi feldolgozó egység), mikroprocesszor. A modern processzor rendkívül összetett, eredeti három fő része, a vezérlő egység, az aritmetikai és logikai egység (Arithmetic Logic Unit – ALU) és a regisztertömb, számos további résszel egészült ki. Ezek közül csak a cache-t említjük.

- Vezérlő egység (Control Unit – CU): Feladata az adatátvitel vezérlése a processzor és a perifériák között, beleértve a memóriát is. Irányítja a buszokat, beolvassa az utasításokat és adatokat, kiírja az eredményeket.
 - Beolvassa az utasításregiszterbe (IR: Instruction Register) a következő utasítást, értelmezi a kódját, ennek megfelelően vezérli az ALU-t, olvas be adatokat, kezeli a vermet és az ugrásokat. Az utasításszámláló regiszterben (PC: Program Counter) tárolja a következő utasítás címét. Normál esetben az utasítás végrehajtását követően egy utasítást lép előre, ugró utasításnál pedig a megfelelő címre változik értéke.
- Aritmetikai és logikai egység (Arithmetic Logic Unit – ALU): Aritmetikai műveletek: +, -, *, /, %, Logikai művelet: <, >, <=, >=, !=, &, ^, ~, ~. A lebegőpontos műveleteket külön egység végzi, az FPU (Floating Point Unit).
- Regiszterek: Általában 8-128 bit széles írható-olvasható tárolók, néhány tucat található belőlük a processzorban, legtöbbször egyedi neve van (pl. IR, PC, SP, RAX, stb.). Architektúrától függően néhány fix méretű regisztertömb is lehet a processzorban (xmm0, xmm1, stb.). Az ALU által használt regisztereket akkumulátornak nevezük. **Általában a „Hány bites a számítógép?” kérdésre az akkumulátor mérete a válasz.**
- Cache: a RAM messze van a processzortól, tehát nagy a késleltetési idő, ha a gigahertzes órajeleket vesszük figyelembe. Ráadásul a DRAM eleve lassú. Ezért a processzor nem csak az éppen szükséges adatot, hanem a következő várhatóan szükséges adatokat is beolvassa az átmeneti tárolóba, a cache-be. Gyakori, hogy ugyanaz a programkód vagy adat sokszor kerül lefuttatásra vagy feldolgozásra. Ha ezeket a CPU a cache-ben tartja, nem kell várakoznia a RAM-ra. Az operációs rendszer a memóriát használja a háttértár adatainak cache-elésére, így tehát a cache szót tágabb értelemben is használjuk.

Utasításkészlet típusok ►

- CISC (Complex Instruction Set Computing – összetett utasításkészletű feldolgozás): Az utasítások között számos összetett műveletet megvalósító is található, pl. sztringfeldolgozás. Ezeket a processzor nem közvetlenül hajtja végre, hanem egyszerűbb utasításokra bontja. Az utasítások nem egyforma méretűek, ezért nem lehet előre tudni, hol kezdődik a 10. következő utasítás, csak ha az előző 9-et beolvastuk. Sokféle címzési mód támogatott. Mivel az x86-os architektúra, mely CISC felépítésű, rendkívül elterjedt, a processzorgyártók olyan mértékben optimalizálták a processzoraikat, hogy azok sebességben felveszik a versenyt az elvileg hatékonyabb RISC és VLIW architektúrákkal. Ezt úgy érték el, hogy a beérkező utasításokat RISC jellegű utasításokká alakítják, és igen fejlett elágazásbecslést, valamint a kódban szereplő sorrendtől eltérő végrehajtási sorrendet (Out of Order Execution) alkalmaznak. Kihasználják a CISC rendszernek azt az előnyét, hogy mivel az utasítások összetettek, kevesebb kell belőlük ugyanahhoz a feladathoz, tehát kevesebbszer kell olvasni a memóriából.
- RISC (Reduced Instruction Set Computing – csökkentett utasításkészletű feldolgozás): Az utasítások egyszerűek és egyforma méretűek, a processzor közvetlenül végrehajtja őket. Régen a CISC-nél sokkal magasabb órajelek elérését tette lehetővé, ma már a CISC optimalizálásai miatt nem így van.
- VLIW (Very Long Instruction Word – nagyon hosszú utasításszó): Fix nagyméretű utasításegységekből épül fel a kód, az egy egységen belül található utasításokat a processzor párhuzamosan hajtja végre. Nem változtatják meg az utasítások sorrendjét, az optimalizálás teljes mértékben a fordítóra marad, éppen ezért sokkal erősebben igényli a hatékony fordítót, mint a CISC, RISC rendszerek.
 - EPIC (Explicitly Parallel Instruction Computing – közvetlenül párhuzamos utasításkészletű feldolgozás): a VLIW Intel által továbbfejlesztett verziója, IA64

Operációs rendszerek ► Számos olyan művelet van, amit sok program igényel, azonban nem érdemes hardveresen megvalósítani.

Az operációs rendszer fő feladata, hogy biztosítsa ezeket a műveleteket a programok számára.

Például az stdio.h függvénykönyvtár függvényeinek zöme feladatát az operációs rendszer által biztosított függvények meghívásával valósítja meg.

- A printf függvény úgy működik, hogy a printf függvény létrehoz egy sztringet a képernyőre írandó tartalommal, és ezt a sztringet átadja az operációs rendszernek. Az operációs rendszer jeleníti meg a karaktereket a képernyő megfelelő részén. Ezért lehet például, hogy az egyik operációs rendszerben jól jelennek meg a magyar karakterek, a másikban pedig rosszul. (A %d behelyettesítést a printf végzi.)
- A scanf és egyéb beolvasások egy memóriaterület címét adják át az operációs rendszernek, ahová az beolvassa a felhasználó által begépelte adatokat. Az operációs rendszer nem tudja, hogy scanf, getchar, gets vagy mi egyéb hívta a függvényét, ezért mindig az ENTER lenyomásáig olvas, és ezután kerül vissza a vezérlés a programhoz, ahol a beolvasott adatok értelmezése és feldolgozása megtörténik.
- A különféle erőforrások (fájlok, dinamikus memória) kezelése is az operációs rendszer dolga.

Az operációs rendszerek készítői nem ismerhetnek minden hardvert, amit a felhasználók csatlakoztatnak a számítógéphez, ezért a hardverek használatához szükséges függvényeket a hardverek gyártóinak kell biztosítani (driver). Az operációs rendszer az interfészt, azaz a függvények deklarációját adja meg, melyekhez illeszkednek a driverek függvényei.

A lefordított programról ► PC-n, 32 bites Windowsban nézünk egy példát.

```
#include <stdio.h>

int minad(int a, int b, int c){
    int res;
    if(a>b){
        if(a>c)res=b+c;
        else res=a+b;
    }
    else if(b>c)res=a+c;
    else res=a+b;
    if(res>a && a<0)res=-res;
    return res;
}

int main(){
    int x=minad(10,20,50);
    printf("Eredmeny=%d\n",x);
    return 0;
}
```

Fordítóból kinyerhető gépi kód+assembly+forráskód►

A printf formátumsztring konstansa

```
CONST SEGMENT
??_C@_0N@NIDIBBAO@Eredmeny?%DN?%CFd?6?%AA@ DB
'Eredmeny=%d', 0aH, 00H ;
CONST ENDS
```

A kék az automatikusan generált változónév. DB=Define Byte (a karaktereket egy bajtosnak fordítja, ez fordítófüggő!). Aposztrófok között az assembly formátumú konstans sztring, mely nem tartalmazza az újsor és a lezáró 0 karaktereket, ezért ezek utána vannak felsorolva: 0aH a hexadecimális 13, ami a \n kódja, és a 00H, ami a 0 (más néven '\0') kódja.

A main fordításának eredménye

```
; 16 : int x=minad(10,20,50);
0001e 6a 32 push 50 ; 00000032H
00020 6a 14 push 20 ; 00000014H
00022 6a 0a push 10 ; 0000000aH
00024 e8 00 00 00 00 call ?minad@YAHHHH@Z ; minad
00029 83 c4 0c add esp, 12 ; 0000000cH
0002c 89 45 f8 mov DWORD PTR _x$[ebp], eax

; 17 : printf("Eredmeny=%d\n",x);
0002f 8b 45 f8 mov eax, DWORD PTR _x$[ebp]
00032 50 push eax
00033 68 00 00 00 00 push OFFSET
FLAT:??_C@_0N@NIDIBBAO@Eredmeny?%DN?%CFd?6?%AA@
00038 e8 00 00 00 00 call _printf
0003d 83 c4 08 add esp, 8

; 18 : return 0;
00040 33 c0 xor eax, eax
...
00055 c3 ret 0
```

Piros: Az utána következő gépi kód memóriacíme a main függvény kezdőcímehez képest, hexadecimális formában megadva. A piros számok természetesen nincsenek benne a lefordított kódban, csak az olvashatóságot segítik.

Zöld: Ezek a tényleges gépi kódok hexadecimális alakban, vagyis azok a bájtok, amik megtalálhatók az exe fájlban, ha megnyitjuk, és végigolvassuk. Minden sorban egy gépi kódú utasítás szerepel. A függvények és a sztring címe nincs behelyettesítve. Felismerhetjük, hogy CISC architektúráról van szó, hiszen az utasítások hossza különböző. (32 bites x86 kód egyébként.)

Kék: Az assembly kód, ez a gépi kód ember számára olvashatóbb változata.

- A push utasítás a verembe menti az értékeket (erről lesz szó a félév későbbi részén).
 - Érdekeség: Figyeljük meg, hogy fordított sorrendben kerülnek az adatok a verembe, emiatt a verem tetejére kerül a legelső paraméter. C-ben vannak változó paraméterű függvények, amilyen a printf vagy a scanf, ezeknél csak az első paraméter biztos, a többi paramétert a formátumsztring alapján azonosítja a meghívott függvényt. Ha fordítva kerülne a verembe az értékek, akkor nem tudná, hol keresse az első paramétert. Más programnyelvek, pl. a Pascal nem, így működnek, azok sorban teszik el a paramétereket. Ha egy C program Pascal módon írt függvényt kell hívjon, pl. az operációs rendszer egy függvényét, akkor a függvényt Pascal típusúnak kell deklarálni. Pl.: pascal int fuggveny(int x, int y); Ilyesmiről nem lesz szó ebben a félévben.
- call a függvényhívás (a paraméterek a verem kerülnek átadásra).
- add esp, 12: 12-t ad az esp regiszterhez, azaz felszabadítja a paraméterek által lefoglalt helyet a veremben. Az esp az Extended Stack Pointer rövidítése, ez a 32 bites veremmutató. Azért ad hozzá 12-t, mert a push utasítással beletett 3 db 4 bájtos egész számot, a függvényparamétereket. Ezeket kivethetné pop-pal, de nincs szüksége a paraméterekre (azokat a minad függvény már használta), és így gyorsabb. Azért hozzáad és nem elvesz, mert az x86-os kódban a verem „fejlel felé”, áll, tehát minél több adatot teszünk bele, annál kisebb az esp értéke.
- mov: másoló utasítás, jobbról balra másol. mov DWORD PTR _x\$[ebp], eax: az eax akkumulátor regiszter tartalmát az 'x' változóba teszi. A függvények az eax regiszterben adják vissza az egész típusú visszatérési értéket.
 - A következő sorban az x-ből kerül vissza az adat eax-be. Ez nem azért van, mert a mov átmozgatta, tehát vissza kell tenni, hanem mert debug üzemmódban lett lefordítva a program, és ekkor a fordító semmit sem optimalizál, minden C kódot megvalósít, tehát nem törődik azzal, hogy felesleges műveletet végez.
- A printf hívás előtti push-nál felismerhetjük a formátumsztring memóriacímet.
- xor eax,eax: Ha valamit magával hozunk xor kapcsolatba, akkor 0-t kapunk.
- Assemblyben minden aritmetikai és logikai művelet esetében az első paraméter csak regiszter lehet, és az eredmény az első paraméterbe kerül. Jelen esetben a C nyelv ^= műveletét valósítja meg, korábban pedig az add a +=-t.
- ret 0: Itt a 0 nem a visszaadott érték, ez abból is látszik, hogy a kód mindössze egy bájttal. A visszaadott érték az eax-be kerül.

A minad függvény:

```

; 4 : int res;
; 5 : if(a>b){

0001e 8b 45 08 mov eax, DWORD PTR _a$[ebp]
00021 3b 45 0c cmp eax, DWORD PTR _b$[ebp]
00024 7e 1e jle SHORT $L612

; 6 : if(a>c) res=b+c;

00026 8b 45 08 mov eax, DWORD PTR _a$[ebp]
00029 3b 45 10 cmp eax, DWORD PTR _c$[ebp]
0002c 7e 0b jle SHORT $L613
0002e 8b 45 0c mov eax, DWORD PTR _b$[ebp]
00031 03 45 10 add eax, DWORD PTR _c$[ebp]
00034 89 45 f8 mov DWORD PTR _res$[ebp], eax

; 7 : else res=a+b;

00037 eb 09 jmp SHORT $L614
$L613:
00039 8b 45 08 mov eax, DWORD PTR _a$[ebp]
0003c 03 45 0c add eax, DWORD PTR _b$[ebp]
0003f 89 45 f8 mov DWORD PTR _res$[ebp], eax
$L614:

; 8 : }
; 9 : else if(b>c) res=a+c;

00042 eb 1c jmp SHORT $L615
$L612:
00044 8b 45 0c mov eax, DWORD PTR _b$[ebp]
00047 3b 45 10 cmp eax, DWORD PTR _c$[ebp]
0004a 7e 0b jle SHORT $L616
0004c 8b 45 08 mov eax, DWORD PTR _a$[ebp]
0004f 03 45 10 add eax, DWORD PTR _c$[ebp]
00052 89 45 f8 mov DWORD PTR _res$[ebp], eax

; 10 : else res=a+b;

```

```

00055 eb 09 jmp SHORT $L615
$L616:
00057 8b 45 08 mov eax, DWORD PTR _a$[ebp]
0005a 03 45 0c add eax, DWORD PTR _b$[ebp]
0005d 89 45 f8 mov DWORD PTR _res$[ebp], eax
$L615:

; 11 : if(res>a && a<0) res=-res;

00060 8b 45 f8 mov eax, DWORD PTR _res$[ebp]
00063 3b 45 08 cmp eax, DWORD PTR _a$[ebp]
00066 7e 0e jle SHORT $L618
00068 83 7d 08 00 cmp DWORD PTR _a$[ebp], 0
0006c 7d 08 jge SHORT $L618
0006e 8b 45 f8 mov eax, DWORD PTR _res$[ebp]
00071 f7 d8 neg eax
00073 89 45 f8 mov DWORD PTR _res$[ebp], eax
$L618:

; 12 : return res;

00076 8b 45 f8 mov eax, DWORD PTR _res$[ebp]
...
00055 c3 ret 0

```

- cmp: összehasonlít két egész értéket, az eredmény egy speciális flag regiszterbe kerül. A flagek 1 bites értékek. A cmp egyik paramétere konstans vagy regiszter, ezért kellett előbb bemásolni a-t eax-be.
- jle: feltételes ugrás (Jump if Lower or Equal, ugorj, ha kisebb vagy egyenlő). Ha a flag regiszter tartalma azt jelzi, hogy az összehasonlítás eredménye <= volt, akkor az else ágra ugrik. A jle paramétere az ugrási cím. Jelen esetben egy SHORT érték, ami azt jelenti, hogy 1 bájttal adjuk meg, mennyit kell ugrani. Ennek a bájtnak az értéke adódik az IC utasítászámzó regiszter értékéhez.
- jmp: feltétel nélküli ugrás (Jump) a megadott (relatív) címre.
- A visszatérési érték az eax-be kerül.

Összefoglalva

A memóriában az adatok és utasítások bájtok sorozataként jelennek meg, ránézésre nem lehet megmondani, hogy mi micsoda. Ugyanazt a memóriatartalmat utasításként, egész számként vagy lebegőpontos számként kezelve teljesen más eredményt kapunk.

Például a 2 206 009 344 egész szám (hexa: 83 7D 08 00) lebegőpontos számként -7.43592e-037, gépi kódban pedig a cmp DWORD PTR _a\$[ebp], 0 utasítás kódja.

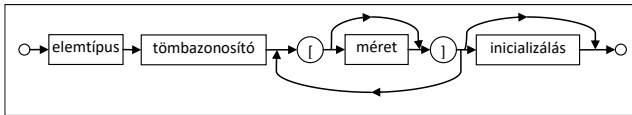
A végrehajtható fájl (pl. exe) eleje fix felépítésű, így indításkor az operációs rendszer tudja, hol van az első utasítás illetve az adatok.

14. Összetett és származtatott adat-típusok, típusdefiníció

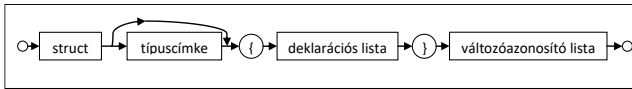
A 10. fejezetben megismerkedtünk a C egyszerű adattípusaival (void, char, egész típusok, lebegőpontos típusok). Vannak azonban olyan adattípusok, melyek ezekből az elemi típusokból vagy más összetett típusokból épülnek fel (tömb, struktúra, union) illetve ezekből származnak (enum, bitmező, pointer, függvény). A C nyelv lehetővé teszi saját adattípusok létrehozását is.

14.1 Összetett és származtatott típusok

Tömb ► A 9. fejezetben már megismerkedtünk ezzel az adattípussal. Egyfajta típusú, szomszédosan elhelyezkedő elemek halmaza a tömb. Egy tömbtípus elemeinek típusa és elemeinek száma határozza meg. A tömbnek legalább 1 eleme kell legyen. A tömb elemei a sorszámuk segítségével érhetők el, a kezdőelem sorszáma 0. **Elemi származtatott illetve összetett típusúak is lehetnek.**



Struktúra ► Tetszőleges típusú, szomszédosan elhelyezkedő elemek halmaza a struktúra. A struktúrának legalább egy eleme kell legyen. Az elemek (tagváltozók) az egyedi nevük segítségével érhetők el. Elemi származtatott típusúak is lehetnek.



A struktúra jól használható például abban az esetben, ha egy függvénynek több értéket kell visszaadnia. Nézzük például a másodfokú egyenlet megoldását:

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

struct par{ double x1,x2; };

struct par gyok(double a,double b,
               double c){
    double d = b * b - 4 * a * c;
    struct par x;

    if(d < 0){
        fprintf(stderr,
            "Negativ diszkriminans.\n");
        exit(1);
    }
    d = sqrt(d);
    x.x1=(-b+d)/(2.0*a);
    x.x2=(-b-d)/(2.0*a);
    return x;
}

int main(){
    struct par x;

    x=gyok(1.0,7.0,10.0);
    printf("a=1, b=7, c=10.\n");
    printf("Az egyenlet gyokei:\n");

```

```

printf("x1=%g\nx2=%g\n",x.x1,x.x2);
return 0;
}

```

A programban a struktúra definíciója így nézett ki:

```

struct par{ double x1,x2; };

```

de gyakran írjuk így, több sorra bontva:

```

struct par{
    double x1,x2;
};

```

Struktúra helyett használhattunk volna kételemű tömböt? Nem. Magyarázat a pointerekkel foglalkozó részben lesz.

Kaphat kezdőértéket: `struct par x={7.1, -4.2};`

A struktúra különböző típusú változókat is magába foglalhat:

```

struct korcs{
    double x1,x2;
    int a;
    char t[10];
};

```

Másolható = operátorral. (Emlékezzünk, a tömb nem! De ha a struktúrában van tömb, akkor a struktúrával együtt másolható! Magyarázat a pointerekkel foglalkozó részben.)

```

struct korcs a,b;
a=b;

```

Függvény átvehet paraméterként struktúrát, és vissza is tudja adni.

```

struct korcs proba(struct korcs a){
    if(a.t[3]=='N')a.x1=a.x2;
    else a.x2=a.x1;
    return a;
}

```

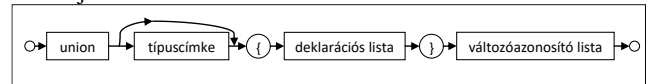
Nem csak struktúrában lehet tömb, tömbben is lehet struktúra:

```

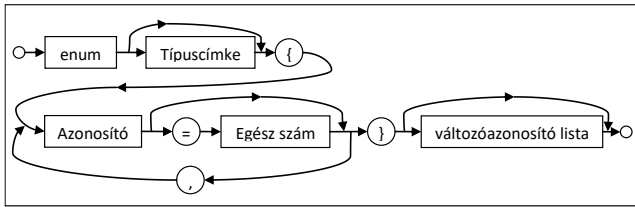
int main(){
    struct korcs kutyu[100];
    int i;
    for(i=0; i<100; i++)kutyu[i].a=0;
}

```

Union ► Tetszőleges típusú, egymást átfedő elemek halmaza a union. A unionnak legalább egy eleme kell legyen. Az elemek (tagváltozók) az egyedi nevük segítségével érhetők el. Elemi származtatott típusúak is lehetnek. Ritkán használjuk. Bővebben a 32. fejezetben szólnunk a unionról.



Felsorolt típus (enum) ► Egész (int) típusú konstansok csoportos definícióját teszi lehetővé.



- Megadás: `enum` azonosító {konstansok};
- Pl.: `enum` lampa{piros,pirossarga,zold,sarga} akt=zold;
- Pl.: `enum` {Zold,Tok,Makk,Piros}szin=makk;
- Pl.: `enum` irany{fel,le,jobbra,balra};
- Pl.: `enum` het{hetfo,kedd,szerda,csutortok=-1, pentek};
- Pl.: `enum` prim{ketto=2,harom,ot=5,tizenhet=17, tizenharom=13, k2=2};

Ha nem adunk meg értéket, a konstansok értéke egyesével nő, az indulóérték pedig 0. A példákban hetfo=0, kedd=1, szerda=2, csutortok=-1, pentek=0. Az enum konstansok azonosítók, tehát nem lehet pl. ilyen nevű változó vagy másik konstans, mert a nevük ütközne.

Példa használatra: Írjunk függvényt, amely egy közlekedési lámpát a következő állapotba vált (piros → pirossarga, pirossarga → zöld, zöld → sarga, sarga → piros). A lámpa állapotát enum-mal adjuk meg!

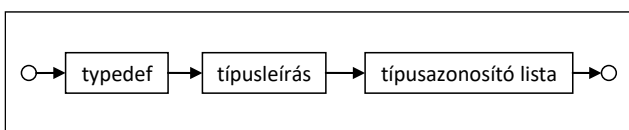
```
#include <stdio.h>
enum lampa {piros, pirossarga, zold, sarga};
enum lampa valt(enum lampa be) {
    switch (be) {
        case piros: return pirossarga;
        case pirossarga: return zold;
        case zold: return sarga;
        case sarga: return piros;
    }
}
```

Bitmező ► Struktúra vagy union int vagy unsigned int típusú tagváltozóiból általunk megadott bitszámú egész értékeket alakíthatunk, ezáltal takarékosabb memóriahasználatot elérve. A megadott bitszám nem lehet nagyobb, mint az adott rendszerben használható int/unsigned mérete. Bővebben lásd a 32. fejezetben.

Pointer vagy mutató ► Egy adott típusú pointer hivatkozást tartalmaz egy, a pointer típusával megegyező típusú elemre. (Azaz a pointer egy változó memóriacímét tartalmazza.) A pointer típusa egyszerű és származtatott típus is lehet. Bővebben lásd a 16. fejezetben.

Függvény ► A függvénytípus meghatározott típusú visszatérési értékű függvényt ír le. A függvénytípust a visszatérési értékének típusa, valamint a paramétereinek száma és típusa írja le. A 28. fejezetben, a függvénypointerek kapcsán világosabbá válik majd a függvénytípus értelme.

14.2 Saját típus létrehozása – a típusdefiniáció



Saját típusnevet a következő módon hozhatunk létre:

```
typedef eredeti_típus új_név;
```

Itt az eredeti típus több részből is állhat, az új név egyszavas C azonosító.

Például:

```
typedef int egesz;
typedef unsigned char uchar;
typedef egesz inttomb[100]; // tömb típus
typedef double dtomb[33]; // tömb típus
typedef struct {
    char nev[100];
    double ertek;
} ember; // struktúra típus
typedef FILE * PFILE; // fájlpontertípus
typedef double (*fvp) (double); // függvénypointertípus
```

```
egesz a=3;
uchar c;
inttomb t1,t2; // 2 db 100 elemű egészeket tároló tömb
PFILE fp=fopen("szoveg.txt","wt");
fvp gyok=sqrt;
double gyokketto=gyok(2);
ember Peter,Pal;
strcpy(Peter.nev,"Peter");
Peter.ertek=3.5;
```

A pointerekről, függvénypointerekről a 16. és 38. fejezetben beszélünk bővebben.

Gondolkozzunk együtt!

Oldja meg egyedül!

15. Operátorok

Ebben a fejezetben összefoglaljuk a C nyelv operátorait, azaz műveleti jeleit, előbb azonban röviden szólnunk a logikai értékek és az egész számok kapcsolatáról. Az operátorokat összefoglaló részben megismételjük a már tanult operátorok legfontosabb tulajdonságait, így az olvasó később mindent megtalál egy helyen.

15.1 Logikai értékek és egész számok

Egész mint logikai érték ► A C nyelvben az egész számoknak logikai jelentése van: ha a szám nem nulla, az IGAZ-at jelent, ha nulla, az HAMIS-at. Ez a tulajdonság kihasználható logikai vizsgálatoknál. Pl.:

```
for(i=100; i>0; i--)...
```

helyett azonos jelentéssel írható, hogy

```
for(i=100; i; i--)... (1)
```

Egy sztringet kiírhatunk karakterenként így:

```
char s[100]="Mndn egér szereti a sajtot";
for(i=0; s[i]!=0; i++) putchar(s[i]);
```

A feltételvizsgálatban szereplő `!=0` helyett írhatjuk, hogy `'\0'`, így:

```
for(i=0; s[i]!='\0'; i++) putchar(s[i]);
```

A két megoldás egyenértékű, mindenki a neki szimpatikusát válassza!

Vizont mivel a `char` is egész típusú (akárcsak az `enum` vagy a `pointer`), itt is használható a rövidebb felírás:

```
for(i=0; s[i]; i++) putchar(s[i]); (2)
```

A logikai értékeken alkalmazható logikai műveletek egészen is alkalmazhatók. Például

```
int a,b;
...
if(a==0 && b!=0)...
```

helyett

```
if(!a && b)...
```

is írható. Ha `a`-ban 0 van, akkor az logikai HAMIS, amiből a `!` operátor logikai IGAZ-at csinál, azaz `!a` akkor igaz, ha `a==0`.

Az `i>0` helyére tehát írható `i`, az `s[i]!=0` helyére írható `s[i]`, az `a==0` helyére írható `!a` és `b!=0` helyére írható `b`, azonban ezek használatát mégis célszerű kerülölni, mert a kódot nehezebben olvashatóvá teszi. Azért szerepel mégis ebben a könyvben, mert a C programozónak fel kell ismernie más kódjában.



A pointerrekről csak a következő fejezetben lesz szó, azonban ide kívánczok, hogy pointerek esetén is gyakran használják ezt a megoldást, melyet azonban pointereknél is célszerű kerülölni. Az `if(!p)...` helyett tehát írjunk `if(p!=NULL)...`-t, az `if(p)...` helyett `if(p!=NULL)`-t, mert így sokkal olvashatóbb a kód. A `NULL` konstans nevét is írjuk ki, ne használjunk helyette 0-t, ugyancsak áttekinthetőségi okból!

Logikai érték mint egész ► A logikai műveletek eredménye egész számként használható. A logikai művelet eredménye 0, ha HAMIS és 1, ha IGAZ. Például:

```
int a=(2>3),b=(2<3);
printf("%d,%d\n",a,b); // 0,1
```

Gyakran használunk olyan függvényeket, melyek logikai értéket adnak vissza. Nézzünk két példát ezek elkészítésére! Az első megvizsgálja, hogy három valós szám lehet-e egy háromszög oldala:

```
int hszog(double a, double b, double c){
    if(a>=b+c) return 0;
    if(b>=a+c) return 0;
    if(c>=a+b) return 0;
    return 1;
}

if(hszog(3.0, 4.0, 5.0))printf("Lehet háromszög");
```

A második eldönti a paraméterként átvett karakterről, hogy betű-e:

```
int betu_e(char ch){
    if(ch>='A' && ch<='Z') return 1;
    if(ch>='a' && ch<='z') return 1;
    return 0;
}

char s[100]="Lenni, vagy nem lenni?";
int n,i;
for(i=n=0; s[i]!=0; i++)if(betu_e(s[i])n++;
printf("%d db betű volt a szövegben.\n",n);
```

A `betu_e` függvény csak az angol ABC betűit vizsgálja. Egészítsük ki úgy, hogy a magyar betűkkel is boldoguljon!

A C nyelv szabványos könyvtárában megtaláljuk azt a függvényt, amely ellenőrzi, hogy egy karakter betű-e. Ennek neve `isalpha`. Használata ugyanaz, mint a `betu_e` függvényé:

```
for(i=n=0; s[i]!=0; i++)
    if(isalpha(s[i])n++;
```

Az `isalpha` mellett további karaktervizsgáló függvények is rendelkezésre állnak, ezek közül a fontosabbak:

- `isalnum`: ha betű vagy számjegy
- `isupper`: ha nagybetű
- `islower`: ha kisbetű
- `isspace`: ha whitespace karakter

Ezek a függvények a `ctype.h` fejléc beépítésével válnak elérhetővé. Ugyancsak a `ctype.h`-ban található két további hasznos függvény, a `toupper` és a `tolower`. Mindkét függvény egy karaktert vár paraméterként, és visszatérési értékük ugyancsak egy karakter: a `tolower` esetében, ha a bemenő karakter nagybetű volt, akkor annak kisbetű változatát adja vissza, minden más esetben az eredeti karaktert kapjuk. A `toupper` esetében pedig a kisbetűt cseréli nagyra.

Például:

```
char s[100]="Mndn egér szereti a sajtot";
for(i=0; s[i]!=0; i++)
    putchar(toupper(s[i]));
```

15.2 A C nyelv operátorai

Az operátorokat kifejezésekben használjuk. Egy operátor jellemzői:

- Milyen OPERANDUSAI vannak
 - hány darab
 - milyen típusú
- Milyen az eredmény típusa/értékkészlete
 - mint az operandusoké
 - eltérő típus (pl. relációs operátorok)
- Operandusok kiértékelési sorrendje
asszociativitás (csopontosíthatóság, pl.: $(a+b)+c \rightarrow a+(b+c)$ ugyanaz (bár számítási pontosság miatt itt is lehet gond, pl. ha b és c egyforma, a pedig hozzájuk képest nagyon kicsi).
- FŐ HATÁS: milyen értéket szolgáltat a kifejezés a kiértékelés után?
- Van-e egyéb hatása a műveletnek \rightarrow MELLÉKHATÁS
- Mi a viszonya a többi operátorhoz egy összetett kifejezésben \rightarrow PRECEDENCIA
- A művelet elvégzése során történik-e valamilyen automatikus típuskonverzió?

Például az értékadás művelete:

- $a = b$
- ez így nem utasítás, hanem egy kétoperandusú művelet \rightarrow eredménye lesz \rightarrow értéke lesz \rightarrow tehát ez egy kifejezés
- általánosan: *balérték* = *jobbérték* (*lvalue* = *rvalue*)
 - Mi lehet jobbérték? Bármilyen, aminek értéke van.
 - Mi lehet balérték? Bármilyen, aminek érték adható (írható adatterületet címez). Pl. változó, pointer kifejezés

Az értékadó operátor működése:

- *rvalue* kiértékelése
- *lvalue* kiértékelése
- az *lvalue=rvalue* kifejezés értéke a kiértékelte *rvalue* értéke lesz \rightarrow ez a FŐ HATÁS
- MELLÉKHATÁSKÉNT *rvalue* értéke az *lvalue* által címzett adatterületre másolódik.
- Következmény: $a=b=c=1$ esetén a kiértékelés a fenti szabályok szerint $c=1$ kiértékelése után már \bar{o} is szolgálhat *rvalue*-ként és így tovább.

Az operátorok többsége nem használható összetett adattípusokon (tömb, struktúra, union), de az ezekben tárolt egyszerű típusú változókon igen. Például:

```
int t1[100], t2[100], t3[100], i;

t1=t2; // HIBÁS
t3=t1+t2; // HIBÁS
for (i=0; i<100; i++) t1[i]=t2[i]; // HELYES
for (i=0; i<100; i++)
    t1[i]=t2[i]+t3[i]; // HELYES
```

A C nyelv operátorait összegző táblázat:

Operátor	Funkció	Példa
++, -- [] () . >	változó növelése/csökkentése 1-gyel(post) tömb indexelése zárójelezés, függvényhívás struktúra tag elérése pointerrel mutatott struktúra tagjának elérése	i++, j-- t[i]=9; printf("Hello világ\n"); ember.sorszam=3; if(gyoker->bal==NULL)
sizeof ++, -- & * +, - ~ ! (típus)	típus vagy változó mérete bajtban* változó növelése/csökkentése 1-gyel(pre) változó címe pointer által mutatott változó előjel bitenkénti negálás logikai NEM típuskonverziós operátor	sizeof(double) ++i, --j scanf("%c",&ch); *p=3; *p+=3; n=-5; a=~b; if(!(a>b))puts("a<=b"); int x=(int)3.14;
*, /, % +, - <<, >>	szorzás, osztás, maradékképzés összeadás, kivonás biteltolás	x=(y*2/3)%5; a=b+c-d; if(1<<4==16)puts("OK");
<, <=, >, >=	logikai műveletek	if(3>2)puts("OK");

Operátor	Funkció	Példa
==, !=	logikai műveletek	if(3!=2)puts("OK");
&	bitenkénti ÉS (AND)	if((x&1)==0)puts("Páros");
^	bitenkénti KIZÁRÓ VAGY (XOR)	if((a^a)!=0)puts("o_o");
	bitenkénti VAGY (OR)	a=a 1;
&&	logikai ÉS	if(ch>='a' && ch<='z')...
	logikai VAGY	if(ch=='N' ch=='n')...
?:	feltételes operátor	max=a>b?a:b;
=, +=, -=, *=, /=, %=, &=, ^=, =, <<=, >>=	Értékadó operátorok	a=3; b+=c; a =1; x<<=1;
,	vessző operátor	c={a=d, j=0, x};

* 1 bajt nem biztos, hogy 8 bit!

A művelet típusa \blacktriangleright Ha egy kétoperandusú operátort eltérő típusú adatokon alkalmazunk, akkor a pontosabb típusú érték pontosabb típusúvá konvertálódik, és a művelet eredménye is pontosabb típusú lesz. Bármelyik lebegőpontos típus pontosabbnak számít bármelyik egész típusnál.

Például $3/2$ eredménye 1, mert két egész osztási eredménye is egész, viszont $3.0/2$, vagy $3/2.0$ eredménye 1.5, mert az egyik operandus double típusú.

Precedencia (kiértékelési sorrend) \blacktriangleright A műveletek „rangja”, erőssége. **A fenti táblázat egy rácsában található műveletek egyforma precedenciájúak.** (Akkor is, ha több sorba vannak írva!) Matematikából jól ismerjük a dolgot.

A kiértékelés iránya \blacktriangleright általában balról jobbra, kivéve az egyoperandusú műveletek, továbbá az értékadások és a feltételes operátor.

Példák a precedencia és kiértékelési irány értelmezésére \blacktriangleright

$a*=b=c+d-e*f+g;$

Sorrend:

```
→ e*f
→ c+d
→ (c+d) - (e*f)
→ ((c+d) - (e*f)) + g
→ b = ((c+d) - (e*f)) + g
→ a* = (b = ((c+d) - (e*f)) + g)
→ (a* = (b = ((c+d) - (e*f)) + g))
```

- Legmagasabb precedenciája a szorzásnak van a kifejezésben, ezért ez hajtódik végre először.
- Az additív műveletek (összeadás, kivonás) egyforma precedenciájúak, ők következnek. Közöttük a kiértékelés iránya dönt, ami balról jobbra történik. Először $c+d$, majd ehhez adódik a szorzat, végül g is.
- Az értékadások maradtak. Ezek jobbról balra értékelődnek ki, ezért előbb b -be kerül a kiszámolt eredmény, majd a szorozódik ezzel, és kerül a -ba.
- Végül azért került zárójelbe az egész, mert ennek is van értéke.

`if (!*p) puts("WoW");`

Egyoperandusú kifejezések: jobbról balra.

Sorrend:

```
→ *p
→ -( *p)
→ !( -( *p))
→ (!( -( *p)) )
```

- p egy pointer, $*p$ az általa mutatott érték.

- vesszük a -1-szeresét
- logikai negálás: ha 0 volt, akkor IGAZ lesz belőle, ha nem 0 volt, akkor HAMIS lesz belőle

`while (*p++) ;`

A `p` pointer kezdetben egy sztring elejére mutat, keressük a sztringet lezáró 0-t. Mivel ++ posztinkremens, ennek a legnagyobb a precedenciája.

Sorrend:

→ `p++`
 → `*(p++)`
 → `*(p++)`

- `p++` utólag növeli meg a `p` pointer értékét, azaz a `*p++` még az eredeti `p` által mutatott értéket adja.
- Ha véget ért a ciklus, akkor `p` a lezáró 0 utáni karakterre mutat, mert az utolsó ++ még megtörténik.
- A műveletek tehát a következők:
 - `p++` jelzi, hogy `p` pointer értékét majd később meg kell növelni.
 - vesszük a `p` által mutatott értéket
 - ha ez 0, akkor a ciklus véget ér, ha nem 0, akkor ismétlődik a ciklus, mindkét esetben végrehajtódik a következő művelet:
 - `p` értéke megnő eggyel
- A ciklus magjában üres utasítás áll.

Mellékhatások kiértékelése ► A C szabvány nem definiálja a mellékhatások kiértékelési sorrendjét, ezért **soha ne alkalmazzunk kétértelmű kifejezést!** Például az `x=i++ + i;` kifejezés értéke nem definiált, fordítófüggő, nem tudhatjuk, hogy a második `i` már megnövelt, vagy még az eredeti!



A ++, --, értékadások, () függvényhívás operátoroknak van mellékhatása.

Léptető operátorok ► ++, --

Égész típusú változókon (ide értve a pointert is) használható, a változó értékét növeli illetve csökkenti eggyel. Két változata van: a preinkremens / predekremens és a posztinkremens / posztdetekremens.

- Pre: `++a` ill. `--b`.
 Pl.: `int a=7,b=3, c; c=++a;` → Az utasítás lefutása után `a` és `c` értéke egyaránt 8 lesz, mert a növelés a `(++a)` értékének felhasználása előtt történik.
- Poszt: `a++` ill. `b--`.
 Pl.: `int a=7,b=3, c; c=a++;` → Az utasítás lefutása után `a` értéke 8, `c` értéke 7 lesz, mert a növelés az `(a++)` értékének felhasználása után történik.

Multiplikatív operátorok ► *, /, %

- szorzás (*): semmi különös
- osztás (/): figyelni kell arra, hogy ha mindkét operandusa egész típusú, akkor az eredmény is egész típusú lesz.

Például `double d=4/5;` eredményeképp `d` értéke 0.0! Ugyanis 4 és 5 egész (ha itt egész típusú változó lenne, ugyanez volna a helyzet), az osztás eredménye egész, azaz 0. Ezt követően az `=` operátor automatikus típuskonverzióval a 0 egész számból 0.0 double értéket hoz létre. Helyesen így írhattuk volna: `double d=4.0/5.0;` illetve elég, ha az egyik operandus valós, pl.: `double d=4.0/5;`. Ha nem konstansokat használunk, akkor explicit típuskonverzió szükséges: `int a=4, b=5; double d=(double)a/b;`. Itt a `(double)` típuskonverziós operátor az

`a`-t konvertálja át, mivel a típuskonverziós operátor precedenciája magasabb, mint az osztás operátoré.

- maradékképzés (%): kizárólag nemnegatív egészeket értelmezett, és a két operandusának osztási maradékát adja.

Két szám akkor osztható egymással, ha osztási maradékuk 0, azaz `a%b==0`.

- `x=(a+b)/c*d` és `x=(a+b)/(c*d)` nem ugyanazt jelenti, hiszen a multiplikatív operátorok precedenciája azonos, az első kifejezés tehát így is írható: `x=((a+b)/c)*d`

Logikai operátorok ► &&, ||, !

- A logikai operátorok igazságtáblázatai:

A	B	A&&B	A	B	A B	A	!A
HAMIS	HAMIS	HAMIS	HAMIS	HAMIS	HAMIS	HAMIS	IGAZ
IGAZ	HAMIS	HAMIS	IGAZ	HAMIS	IGAZ	HAMIS	IGAZ
HAMIS	IGAZ	HAMIS	HAMIS	IGAZ	IGAZ	IGAZ	HAMIS
IGAZ	IGAZ	IGAZ	IGAZ	IGAZ	IGAZ	IGAZ	HAMIS

- De Morgan azonosságok: `!(A&&B) == !(A)||B`, ill. `!(A)||B == !(A&&B)`. Ellenőrizze!
- A ill. B a leggyakrabban valamely összehasonlító operátorral (`==`, `!=`, `<`, `<=`, `>`, `>=`) megadott kifejezés.
- A C nyelvben az egész számok logikai értéként kezelhetők: `0==HAMIS`, `1==IGAZ`. Pl.: `1&&3` IGAZ, `0&&3` HAMIS, `0||3` IGAZ, `0||0` HAMIS, `!0` IGAZ, `!1` HAMIS.
- A C nyelvben a logikai műveletek eredménye egész számként kezelhető. A logikai művelet mindig 0-t vagy 1-et ad. Pl.

```
int x;
x=3>1;      →      x==1
x=3<1;      →      x==0
x=3&&1;     →      x==1
```

```
Mit ír ki?
int a=3,b=6;
printf("%d\t%d\t%d\t%d\n",a&&b,a||b,!a,!a);
```

- Az `&&` és a `||` operátor **rövidzár** tulajdonsággal bír. Ez azt jelenti, hogy ha az operátor bal oldalán álló kifejezésből egyértelmű, hogy mi a logikai érték, akkor a jobb oldal nem értékelődik ki. **SOHA NE LEGYENEK MELLÉKAHTÁSSAL BÍRÓ ÖSSZETETT KIFEJEZÉSEK AZ OPERANDUSOK!!**
 - `A||B` esetén ha `A==IGAZ`, nem számít B értéke, az eredmény IGAZ.
 - `A&&B` esetén ha `A==HAMIS`, nem számít B értéke, az eredmény HAMIS.
 - Pl.: `if((x=a/2)&&(y=b/2))...` utasításban ha `a` értéke 0, akkor `y`-ba nem kerül bele `b/2`.
 - Hasznos pl.: `if(a!=0 && b/a>3)...`, vagy `if(p!=NULL && p->ertek<7.1)...` esetben.
- A rövidzár tulajdonság miatt az `&&` ill. `||` operátorok **kiértékelési pontot** (sequence point) is jelentenek, lásd később!



Bitműveletek ► &, |, ^, ~, <<, >>

Égész típusú értékeken értelmezettek, a számok kettes számrendszerbeli alakjával dolgozunk.

- A bitenkénti logikai operátorok igazságtáblázatai:

A	B	A&B	A	B	A B	A	B	A^B	A	~A
0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	1	1	0	1	0	1
0	1	0	0	1	1	0	1	1	1	0
1	1	1	1	1	1	1	1	0	1	0

- A bitenkénti operátorok a számok azonos helyiértékű bitjei között működnek, a számok minden egyes bitjére.

- o Pl.: $11 \& 7 \rightarrow 001011 \& 000111 = 000011 \rightarrow 3$, azaz $11 \& 7 == 3$.
- o Pl.: $11 | 7 \rightarrow 001011 | 000111 = 001111 \rightarrow 15$, azaz $11 | 7 == 15$.
- o Pl.: $11 \wedge 7 \rightarrow 001011 \wedge 000111 = 001100 \rightarrow 12$, azaz $11 \wedge 7 == 12$.
- o Pl.: $\sim 7 \rightarrow \sim 000111 = 111000 \rightarrow -8$, azaz $\sim 7 == -8$. (Előjel nélküli egésznél a típus bitszámától függ a végeredmény, mert a szám elején minden 0-ból 1 lesz.)
- A következő azonosság mindig igaz: $(A/B) == (A \& B) + (A \wedge B)$.
- A bitenkénti operátorok precedenciája kisebb az összehasonlító operátoroknál, emiatt az $if(x \& y == 3)$ feltétel jelentése: $if(x \& (y == 3))$, nem pedig $if((x \& y) == 3)$!!
- Biteltolás (shiftelés) operátorok:
 - o $A \gg$ balra tolja a számot, egy bittel való eltolás 2-vel való osztásnak felel meg. Pl.: $19 \gg 2 == 4$. Ha a szám előjel nélküli, vagy előjeles, de nem negatív, akkor 0 bitek lépnek be balról, a jobb oldalon kieső bitek elvesznek. Ha a szám negatív, 1-es bitek lépnek be balról.
 - o $A \ll$ jobbra tolja a számot, egy bittel való eltolás 2-vel való szorzásnak felel meg. Pl.: $19 \ll 2 == 76$. 0 bitek lépnek be jobbról, a bal oldalon kieső bitek elvesznek.
 - o Az egyik oldalon kieső bitek nem jönnek vissza a másik oldalon.
 - o $x \ll y$ művelet nem változtatja meg x értékét. Ha változást akarunk, pl. így kell: $x \ll= y$.



- o $\|, \&\&: ((k=i++) \|\ (j=-i)) \rightarrow k$ -ba i eredeti értéke kerül mindenképp. Ha i eredeti értéke 0, akkor j -be -1 kerül, mert a $++$ növelés a bal oldal kiértékelése után, a jobb oldal kiértékelése előtt történik. Ha i eredeti értéke nem 0, akkor biztos, hogy a $\|$ művelet eredménye IGAZ, tehát a rövidzár miatt a $j=-i$ kifejezés nem értékelődik ki, azaz j értéke nem változik.
- o $?: x = i++ ? i+1 : i-1; \rightarrow$ a feltétel IGAZ vagy HAMIS volta i eredeti értéke alapján dől el, i értéke ezután megnő, és a kifejezés értéke a meghatározott feltételnek megfelelően $i+1$ -et vagy $i-1$ -et ad vissza. Ha i kezdetben 0, akkor x -be 0 kerül, egyéb esetben pedig az eredeti $i+2$.

Gondolkozzunk együtt!

Oldja meg egyedül!

Mit ír ki?

```
int a=3,b=6;
printf("%d\t%d\t%d\t%d\n", a&b, a|b, a^b, ~a);
```

Feltételes operátor ► ?:

Két részből áll, három operandusa van: *feltétel-kifejezés* ? *igaz-kifejezés* : *hamis-kifejezés*

Ha a feltétel IGAZ, az operátor *igaz-kifejezés* értékét adja vissza, ha HAMIS, *hamis-kifejezés* értékét.

Az $if(a > b) c = a; else c = b;$ utasítás így írható egyszerűbben: $c = a > b ? a : b;$

Vessző operátor ► A vesszőt (,) általában nem operátorként használjuk, például a változódefiníciónál ($int\ x,y;$) nem operátor, a függvényhívásnál ($fgets(s,100,fp);$) nem operátor, ezekben az esetekben nem garantált a balról jobbra kiértékelés, és nem is kiértékelési pont a vessző.

Leggyakrabban a for ciklus fejében használjuk operátorként:

```
for (i=0, j=1; i<n; i++, j<=1)
    printf("2 %u. hatványa=%u\n", i, j);
```

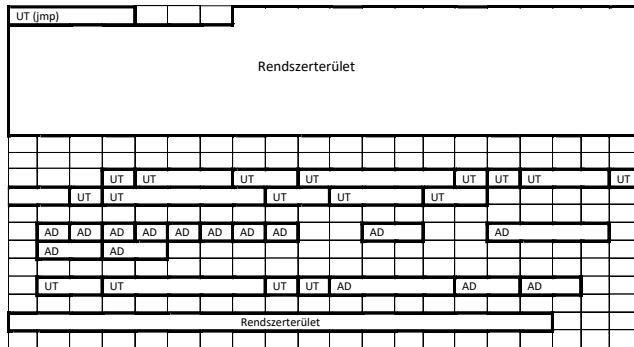
A vessző a legkisebb precedenciájú operátor, még az értékadásnál is kisebb. Mint minden operátornak, ennek is van értéke: a jobb oldali kifejezés értéke. Pl.: $x=(3,4)$; x -be 4 kerül. Pl.: $x=(i=0,j=1)$; itt x -be 1 kerül. A vessző operátor kiértékelési pont.

Kiértékelési pont (sequence point) ► Mikor kerülnek kiértékelésre a kifejezések?

- Utasítás végén: a $;$ -nél vagy $\}$ -nél.
- Függvényhívás előtt az összes paraméter kiértékelődik.
- $if, while, switch ()$ között.
- Néhány operátornál megtörténik a kiértékelés, azaz ezek használatakor garantált a mellékhatások megtörténte az operátor kiértékelése közben:
 - o vessző: $(i=j+2,k=i) \rightarrow k$ -ba garantáltan $j+2$ kerül.

16. Pointerek

16.1 A számítógép memóriája



A memóriát tekintjük egy egydimenziós tömbnek, melynek az egyes bájtoit a *MEMÓRIA[i]* módon indexelve érhetjük el (nincs ilyen tömb a C-ben, a pointerok azonban pontosan így viselkednek).

A fenti ábra abban segít, hogy el tudjuk képzelni egy számítógép memóriájának tartalmát. Korábban már láttunk gépi kódot, ez alapján nem nehéz felismerni az utasításokat reprezentáló UT cellákat. Az AD cellák adatokat (változók, konstansok) jelentenek. A Rendszerterület lehet egyrészt az operációs rendszer adatait és utasításait tároló memóriarész, másrészt a rendes memória címtartományában más, memória jellegű perifériák is megjelenhetnek, így például a videokártya memóriája közvetlenül címezhető ebben a címtartományban, pedig az valójában a videokártyán található. (Ez az oka annak, hogy 32 bites Windows alatt max. 3 GB memóriát lát az operációs rendszer, a maradék 1 GB a videokártya memóriájának címtartománya).

A memóriaterkép egy-egy cellája egy-egy bájtot reprezentál, bár a valóságban általában sokkal több utasítás vagy adat található együtt. Látható, hogy nincs minden területen adat vagy utasítás, ezekkel a területekkel részben a programunk, részben az operációs rendszer gazdálkodik. Például dinamikus memória-foglalás esetén az operációs rendszer ebből ad annyi bájtot, amennyit kérünk tőle (már persze ha rendelkezésére áll a kívánt mennyiség).

A memória elején gyakran ROM található, ahol az első utasítás általában egy ugrás a rendszerindító programra.

16.2 A pointer

Memóriacím egy egész szám, amely kijelöli a memória valahányadik báját.

Pointer (mutató) olyan változó, amelyik memóriacímet tartalmaz.

Pointer típusa: amilyen típusú adatra vagy függvényre mutat a pointer.

Pointer létrehozása az indirekció operátorral (*) történik.

int-re mutató típusos pointer: `int * p;`
A * bárhová írható: `int* p;` `int *p;` `int*p;`

Típus nélküli pointer: `void * p;` (ejtsd: void csillag).

NULL pointer: a memória 0. bájátára mutat. A memória 0. bájta nem tartozik a programhoz, ezért ott nem található a program kódja vagy adata. Egy pointer értékét akkor állítjuk NULL-ra, ha jelezni akarjuk, hogy a pointer nincs beállítva. A NULL

egy konstans, ha használni szeretnénk, a programba be kell szerkeszteni valamelyik szabványos fejlécfájlt (pl. `stdio.h`, `stdlib.h`).

Függvénypointer: egy függvény első utasításának címére mutat. Később bővebben is foglalkozunk vele. Ha nem hangsúlyozzuk, akkor a továbbiakban adatpointerekről lesz szó.

Változó memóriacíme: A fenti ábrán láthatjuk, hogy minden adat van valahol a memóriában. A változó nevéből a címképző (&) operátorral kaphatjuk meg a változó címét:

```
int x, *p;  
p=&x;
```

Pointer által mutatott érték: Indirekció operátorral kapjuk. Ha p a pointer *p az általa mutatott értéket jelenti.

```
int n, *p;  
p=&n;  
// Az alábbi két értékadás ugyanaz  
n=8;  
*p=8;
```

Kezdőérték pointernek: `int n, *p=&n;`. Itt látszólag a pointer által mutatott érték kap értéket, de nem ez a helyzet: a pointernek így adhatunk kezdőértéket, amikor definiáljuk.

Tömb és pointer: Egy tömb neve az index operátor ([]) nélkül egy pointer a tömb első (azaz 0 indexű) elemére.

```
int t[100], *p;  
// Az alábbi két értékadás ugyanaz  
p=t;  
p=&t[0];  
// Az alábbi két értékadás ugyanaz  
p=t+5;  
p=&t[5];
```

Pointer mint tömb: Pointerre alkalmazva az index operátort tömbként kezelhetjük a mutatott memóriát. Fontos: ha olyan memóriát akarunk használni, ahol nincs ténylegesen adat, futás-idejű programhibát kapunk. Az ilyen hibák felderítése nem könnyű.

```
int t[100], *p=t, n;  
  
// Az alábbi két értékadás ugyanaz  
  
t[6]=9;  
p[6]=9;  
  
// Az alábbi két értékadás ugyanaz  
  
p=t+5;  
p=&t[5];  
  
// Az alábbi két értékadás ugyanaz  
// (p a tömb 5-ös indexű elemére mutat)  
  
t[6]=9;  
p[1]=9;  
  
// egy int címét kapja  
  
p=&n;  
  
// Az alábbi három értékadás ugyanaz  
  
n=8;  
*p=8;  
p[0]=8;
```

```
// A következő értékadás hibás:

p[1]=8; // HIBÁS! p 1 db int értékre
        // mutat, az utána következő
        // memóriahely már más adatot
        // vagy utasítást tartalmaz,
        // valamit fölülírunk, mit nem
        // lenne szabad.
```

Gondolkozzunk együtt!

Oldja meg egyedül!

Több pointer definiálása: Tipikus hiba: `int* a, b;`

Itt csak *a* lesz pointer, *b* egy int típusú változó. Helyesen így kellett volna megadni:

```
int *a, *b; // Persze az int* a, *b; is
           // helyes, csak becsapós.
```

Természetesen ez a megoldás is használható:

```
typedef int * pint;
pint a, b;
```

16.3 Pointeraritmetika

Mivel a pointerok előjel nélküli egész számokat tárolnak (a memóriacímét), logikus, hogy matematikai műveleteket is lehessen végezni ezekkel. A pointerok sajátosságai miatt azonban csak néhány ilyen művelet megengedett.

- Pointer és egész szám összeadása vagy kivonása:

```
int t[20], *p=t+10;
*(p-2)=6; // azaz t[8]=6;
*(p+3)=6; // azaz t[13]=6;
```

- **FONTOS!** A pointerhez adott egész szám nem közvetlenül adódik a pointerben tárolt memóriacímhez, hanem úgy, hogy a pointert tömbcímnek tekintve az egész számban meghatározott indexű tömbelem címét kapjuk. Pl.:



```
int t[20], *p=t+10;
printf("p=%u, p+1=%u, p+2=%u\n", p, p+1, p+2);
```

A kiírás eredménye adott gépen (a konkrét számok mindenkinél mások lehetnek): `p=1244792, p+1=1244796, p+2=1244800`. Azaz a memóriacím az int bájtban kifejezett méretével nőtt (int* pointerről van szó), jelen esetben 4-gyel, miközben 1-gyel nőtt a pointerhez adott egész szám.

- Pointer növelése vagy csökkentése ++ ill. – operátorral.

```
int t[20], *p, i;
// A következő két sor ugyanazt teszi
for (p=t; p<t+20; p++) *p=0;
for (i=0; i<20; i++) t[i]=0;
```

A két megoldás közül a tömböset (azaz a másodikat) ajánlott használni, mert

- 1) ez áttekinthetőbb kódot eredményez
- 2) A fordító jobban tudja optimalizálni, hatékonyabb kódot készít.

- Két pointer különbsége

```
int t[20], *p, *q;
p=t+3, q=t+17;
printf("q-p==%d, t-p=%d\n", q-p, t-p);
Eredmény: q-p==14, t-p=-3
```

17. Érték és cím szerinti paraméterátadás; tömbök, pointerok és függvények

17.1 Paraméterátadás

Mit ír ki az alábbi program?

```
#include <stdio.h>

void csere(int a,int b){
    int c=a;
    a=b;
    b=c;
}

int main(void){
    int x=2,y=5;
    csere(x,y);
    printf("x=%d, y=%d\n",x,y);
    return 0;
}
```

Azt írja ki, hogy $x=2$, $y=5$. Miért, hiszen a *csere* függvény megcseréli az értékeket, nem?

Nem. Emlékezzünk vissza, mit mondtunk a függvények paramétereiről! A paraméterként definiált változók (a,b) mindössze abban különböznek az egyszerű változóktól (c,x,y), hogy kezdőértékként kapják azt az értéket, amivel a függvényt meghívjuk. Vagyis a -ba bekerül 2, b -be pedig 5. A függvény helyesen működik, és valóban felcseréli a és b értékét, azonban ez semmi módon nem hat vissza x -re és y -ra.

Ezt úgy mondjuk, hogy a C nyelvben **érték szerinti paraméterátadás** történik.

Tudunk-e írni mégis olyan függvényt, amely felcserél két értéket?

Igen, most, hogy tanultuk a pointereket, már tudunk. Ha ugyanis a függvény nem a változót, hanem annak címét kapja paraméterként, akkor a cím másolódik be a paraméterként definiált pointer változóba, a cím pedig továbbra is az eredeti változóra mutat. Igaz, hogy az eredeti változó a hívó függvényben van, de ez minket egyáltalán nem zavar, jogunk van megváltoztatni annak értékét.

Nézzük a módosított programot!

```
#include <stdio.h>

void csere(int * a,int * b){
    int c=*a;
    *a=*b;
    *b=c;
}

int main(void){
    int x=2,y=5;
    csere(&x,&y);
    printf("x=%d, y=%d\n",x,y);
    return 0;
}
```

Ez már valóban felcseréli x és y értékét, azaz $x=5$, $y=2$ íródik ki.

Figyeljük meg a változásokat:

- A függvény fejébe $int * a$ és $int * b$ került, azaz a függvény pointereket vesz át, a és b pedig int-re mutató pointer.

- A függvényben a és b helyett mindenhol $*a$ ill. $*b$ szerepel. Ugyanis most a és b már nem int típusú változók, hanem int-re mutató pointerok, az általuk címzett változót pedig az indirekció operátor segítségével érhetjük el.
- A csere függvény hívása megváltozott: *csere(a,b)* helyére *csere(&a,&b)* került. Az & címképző operátor a két változó címét adja. Ne felejtjük odaírni, ellenkező esetben az a pointerbe 2, a b pointerbe pedig 5 kerül, és a csere függvény a memória 2. és 5. bájtyán lévő int típusúnak tekintett (amúgy ki tudja, hogy valójában mi van ott, adat egyáltalán, vagy utasítás) értéket próbálja megváltoztatni. Ez valószínűleg nem fog sikerülni neki, és elszáll a programunk.

A C-ben így valósítható meg a **cím szerinti paraméterátadás**.

Vegyük észre, hogy a scanf függvénybe pontosan a cím szerinti paraméterátadás miatt kell & jelet írni a változók neve elé, hiszen a scanf megváltoztatja a változók értékét. Ha sztringet olvassunk be, akkor nem írjuk oda az & jelet, mert a sztringet tároló karaktertömb neve a szögletes zárójelek nélkül a tömb kezdőelemének címét jelenti, tehát abban az esetben már eleve rendelkezésre áll a cím. Azonban a következő módon visszacsempészhetjük az & jelet:



```
char s[100];
scanf("%s",s);
```

helyett írhatjuk:

```
scanf("%s",&s[0]);
```

Utóbbit a gyakorlatban nem használjuk, csak érdekességképpen került ide.

Írjon függvényt, mely paraméterként veszi át egy kocka valós típusú értékkel megadott élhosszúságát, és **paramétersoron** visszaadja a kocka felületét és térfogatát!

Tipp: a felület és a térfogat paramétersoros adandó vissza, nem pedig visszatérési értéként (azaz returnnel), tehát a függvénynek három paramétere lesz, egy double érték az élhossznak, és két double típusú pointer: a felületnek és a térfogatnak.

Pointer értékének megváltoztatása függvényben ► Mi a helyzet akkor, ha egy pointert kell visszaadnunk paramétersoron? Ekkor pointerre mutató pointert kap a függvényünk.

Írjunk függvényt, mely paraméterként kap egy egészektől álló tömböt és egy egész értéket, és logikai IGAZ vagy HAMIS értéket ad vissza attól függően, hogy megtalálható-e a tömbben az egész érték. Adja vissza paramétersoron a megtalált elem címét, vagy ha nincs a tömbben, NULL pointert!

```
#include <stdio.h>

int keres(int t[],int n,int mit,int **p){
    int i;
    for(i=0; i<n; i++){
        if(t[i]==mit){
            *p=t+i; // vagy &t[i]
            return 1; // IGAZ
        }
    }
    *p=NULL;
    return 0; // HAMIS
}

int main(void){
    int tomb[]={3,6,3,5,2,3,0,6,-5,92};
```

```

int *cim;
int mit=-1;
scanf("%d",&mit);
if(keres(tomb,10,mit,&cim)
    printf("Megvan, t[%d]==%d\n"
        ,cim-tomb,*cim);
else printf("Nincs meg.\n");
return 0;
}

```

Megjegyzés: a feladat megoldása csak a keres függvény. A `#include`-ot és a `main`-t csak a jobb érthetőség érdekében írtuk ide.

Az `int **p` egy egészekre mutató pointerre mutató pointer. A pointert ebből `*p` módon nyerhetjük ki, `**p`-vel pedig az egész értéket.

A `keres` meghívása után `cim`-be kerül a megtalált elem címe. A `cim-tomb` adja a megtalált elem indexét, de csak akkor, ha a cím a `tomb` valamelyik elemére mutat (ha `keres` IGAZ értéket ad vissza, akkor ez a helyzet).

Ha a feladat úgy szövelt volna, hogy visszatérési értéként adja a megtalált elem címét, vagy ha nincs meg, NULL-t adjon, valamivel egyszerűbb megoldást kapunk:

```

#include <stdio.h>

int * keres2(int t[],int n, int mit){
    int i;
    for(i=0; i<n; i++)
        if(t[i]==mit)
            return t+i; // vagy &t[i]
    return NULL;
}

int main(void){
    int tomb[]={3,6,3,5,2,3,0,6,-5,92};
    int *cim;
    int mit=-1;
    scanf("%d",&mit);
    if(cim=keres2(tomb,10,mit))
        printf("Megvan, t[%d]==%d\n"
            ,cim-tomb,*cim);
    else printf("Nincs meg.\n");
    return 0;
}

```

A fenti programban igyekeztünk rossz példát mutatni, még-hozzá a `main` függvény `if`-jének fejében. A kultúrált, bár még mindig összetett megoldás ez lett volna:

```
if((cim=keres2(tomb,10,mit))!=NULL)...
```

A cím-be bekerül a visszaadott pointer, majd ellenőrizzük, hogy ez NULL-e. A pointer egy egész szám, a NULL pointer pedig a 0 számnak felel meg, ami logikai HAMIS, ezért működik jól az első változat is.

Pointert visszaadó függvény ► Ahogy a példában láttuk, a függvény pointert is vissza tud adni. Nagyon fontos tudnunk azonban, hogy **egy függvényben a változók megsemmisülnek, amikor a függvény futása befejeződik**: a hely felszabadul, a rendszer más változókat tehet ide. Ez azért fontos, mert **TILOS A FÜGGVÉNY LOKÁLIS VÁLTOZÓJÁNAK CÍMÉT VISSZAADNI**.



```

#include <stdio.h>

int * mitilos(int *a,int b){

```

```

int c,t[3];
return a; // Ez rendben van, a egy
        // külső változóra mutat
return &b; // TILOS!
return &c; // TILOS!
return t; // TILOS!
return t+2; // TILOS!
return &t[2]; // TILOS!
}

```

```

int main(void){
    int x,*p;
    p=mitilos(&x,3);
    *p=3; // Emiatt tilos, ami tilos.
    return 0;
}

```

Természetesen a a függvényben lévő `return`-ök közül csak az első fog lefutni, a többi nem, mivel a `return` azonnali kiugrást eredményez a függvényből.

A fenti program szintaktikailag helyes, még a TILOS! sorok is jók, hiszen pl. a `return &c`; visszaadja a `c` változó címét, ami egy egész szám. A program akkor válik hibássá, ha a megsemmisült változóból akarunk olvasni, vagy írni akarjuk, azaz `*p`-t vagy `p[i]`-t írunk. Márpedig nehéz lenne elképzelni olyan életszerű esetet, amikor a függvény pointert ad vissza, de a mutatott címen lévő változóhoz nem akarunk hozzáférni.

17.2 Tömbök átadása függvénynek

Annak, hogy a tömb szögletes zárójellek nélküli neve a tömb kezdőelemének címét jelenti, lett egy érdekes következménye: **a tömbök cím szerint kerülnek átadásra a függvényeknek**.

Írjunk függvényt, amely paraméterként kap egy tömböt, és visszaadja a tömb elemeinek összegét!

```

#include <stdio.h>

double osszegzo(double t[],unsigned n){
    double sum=0.0;
    unsigned i;
    for(i=0;i<n;i++)sum+=t[i];
    return sum;
}

int main(void){
    double tomb[5]={9.1,-1.3,6,.1,-4};
    double ossz=osszegzo(tomb,5);
    printf("ossz=%g\n",ossz);
    return 0;
}

```

A megoldásban azt látjuk, hogy a „paraméterként kap egy tömböt” utasítást úgy értelmeztük, hogy **a tömb mellett az elemszámot is átadjuk**. Ez szükséges, hiszen más módon nem tudjuk kideríteni, hány elemű a tömb. Írhattunk volna olyan függvényt, amely csak az 5 elemű tömbökre működik, akkor nem kellett volna az elemszám, de a feladat nem ezt írta elő.

A paramétersoron ezt látjuk: `double t[]`. A tömböknél azt beszéljük, hogy tilos méret nélküli tömböt definiálni, itt viszont pont ezt látjuk. A dolgot kétféleképpen is meg lehet magyarázni:

1. A függvényparaméterek kezdőértéként kapják azt az értéket, amivel meghívtuk a függvényt, és itt is erről van szó. Márpedig a tömböknél is láttunk olyat, hogy nem adtunk méretet, mert adtunk kezdőértéket, és a fordító megszámlolta az elemek számát, ez alapján hozta létre a tömböt. Ez a **magyarázat azonban téves**, itt ugyanis nem jön létre új tömb a megadott mérettel. Lássuk az igazi magyarázatot:

2. A `double t[]`; definíció pointert hoz létre. Régen a C nyelvben valóban lehetett így pointert létrehozni a paraméterlistán kívül is, de éppen a félrevezető jelölésmód miatt a legtöbb fordító ezt a formátumot már nem engedélyezi. A `t` tehát egy pointer, amely a tömb első elemére mutat, és a pointeroknál látuk, hogy a pointerok mögé indexet írva tömbként használhatók. Az összegez függvényt írhattuk volna így is, teljesen ugyanazt jelenti:



```
double osszegzo(double * t,unsigned n){
    double sum=0.0;
    unsigned i;
    for(i=0;i<n;i++)sum+=t[i];
    return sum;
}
```

A két megoldás közül bármelyiket választhatjuk, de javasolt az indexes megoldás, mert azon jobban látszik, hogy tömböt adunk át, nem egy-egy elemet.

Mivel a tömb címét adjuk át, az elemeiről nem készül másolat. **Ha tehát megváltoztatjuk egy paraméterként kapott tömb elemeit, akkor az eredeti tömböt változtatjuk.**

Írjunk függvényt, amely nullázza a paraméterként átvett, előjel nélküli egészeket tartalmazó tömb összes elemét!

```
void nullaz(unsigned t[],unsigned n){
    unsigned i;
    for(i=0;i<n;i++)t[i]=0;
}
```

A felcserélhetőséget még egy példával igazoljuk. A csere függvény így is kinézhetne:

```
void csere(int * a,int * b){
    int c=a[0];
    a[0]=b[0];
    b[0]=c;
}
```

És persze a paraméterlistán is lehetne `int a[], int b[]`. Ez a megoldás azonban mégsem javasolt, hiszen itt nem tömbökkel dolgozunk, és ahogy sokszor elhangzott már: nagyon fontos, hogy jól olvasható kódot írjunk.

Tömb a struktúrában ► A 14.1 fejezetben láttuk, hogy struktúrában lehet tömb. Mi a helyzet, ha ezt a struktúrát egy függvény kapja paraméterként, vagy függvény adja vissza visszatérési értéként?

A struktúrában lévő tömb átmásolódik, akár paramétról, akár visszatérési értékről van szó, ebben az esetben tehát nem a tömb címe másolódik, hanem a tartalma. Természetesen ez azt is jelenti, hogy a struktúrában átadott tömb megváltoztatása a hívás helyén nem okoz változást.

Az alábbi programban egy háromdimenziós vektor normálását végezzük. A függvény beleír a struktúra tömbjébe, ez azonban nem okoz változást az eredeti adatban:

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

typedef struct {double k[3];} koordinata;

koordinata normal(koordinata be){
    double hossz=0.0;
    int i;
    for(i=0;i<3;i++)
        hossz += be.k[i] * be.k[i];
    hossz=sqrt(hossz);
}
```

```
for(i=0;i<3;i++)
    be.k[i]/=hossz;
return be;
}

int main(void){
    koordinata x={6.0, 3.1, 4.5}, y;
    int i;

    y=normal(x);
    for(i=0; i<3; i++)
        printf("x[%d]=%10g, y[%d]=%10g\n",
            i,x.k[i],i,y.k[i]);
    return 0;
}
```

17.3 Tömb és pointer újra

A tömbök és pointerok kapcsolatáról sokat beszéltünk már, néhány további dologra hívnánk fel a figyelmet:

- A tömbnév „hozzá van rögzítve” a tömbhöz, azaz a tömbnév mint pointer nem változtatható meg:

```
double t1[10],t2[10],*p;
p=t1; // Helyes.
p[5]=3.14; // Helyes.
t2=p; // HIBÁS! T2 mint pointer
// nem változtatható meg.
t2=t1; // HIBÁS! T2 mint pointer
// nem változtatható meg.
t1++; // HIBÁS! T1 mint pointer
// nem változtatható meg.
p++; // Helyes.
```
- Emlékezzünk a sztringeknél tanultakra! Itt látható, miért nem másolható a sztring (vagy bármely tömb) az = operátorral: mert a tömb neve egy rögzített (konstans) pointer, tehát a `t2=t1` másolásnál pointert próbálunk másolni, nem pedig magát a tömböt. Ha a tömb struktúrában van, akkor már nincs gond, hisz a struktúra neve nem pointer, azaz lehet másolni.
- Ha **struktúrában pointer** van, ami egy tömbre mutat, és a struktúrát átmásoljuk = operátorral, akkor csak a pointer másolódik, a tömb nem. Ez logikus, hiszen a fordító nem tudja, hogy mire mutat a pointerünk, mutat-e egyáltalán bármi értelmesre. A tömb nevével nincs ilyen probléma, az mindig a tömbre mutat.



17.4 pointertömb

Pointerekből is hozhatunk létre tömböt. Pl.:

```
int a,b,c,d;
int *t[4]={&a,&b,&c,&d};
*t[1]=5; // azaz b=5
**t=9; // azaz a=9
t[1]=&d;
*t[1]=5; // azaz d=5
```

Sztringpointerek tömbje:

```
char *s[]={ "Hétfő", "Kedd", "Szerda"},ch;

printf("%s\n",s[2]);
s[2]="Vasárnap"; // pointermásolás!!!
printf("%s\n",s[2]);
ch=s[0][3]; // f betű a "Hétfő"-ből
s[0][3]='X'; // TILOS! A "Hétfő"
// sztring konstans
```

```
strcpy(s[1], "Szombat"); // TILOS! A
// "Kedd" sztring konstans
```

A "Hétfő" és a többi sztring konstans csak olvasható, beleírni ezekben nem szabad, általában a program elszáll, ha megkíséreljük. A rá mutató pointer megváltoztatható, így került oda a "Vasárnap".

Ebben az esetben minden sztringhez akkora memórafogyasztás tartozhat, amekkorát a sztring igényel. A kétdimenziós karaktertömb hasonló a pointertömbhöz, de fontos különbségek vannak:

```
char s[][20]={"Hétfő", "Kedd", "Szerda"}, ch;
printf("%s\n", s[2]);
s[2]="Vasárnap"; // TILOS! s[2] konstans
// pointer
printf("%s\n", s[2]);
ch=s[0][3]; // f betű a "Hétfő"-ből
s[0][3]='X'; // Helyes.
strcpy(s[1], "Szombat"); // Helyes.
```

Minden sztringhez egy-egy 20 elemű karaktertömb tartozik, tehát több helyet foglalunk, mint ehhez a tartalomhoz szükséges, viszont a tartalom megváltoztatható.

Gondolkozzunk együtt!

Oldja meg egyedül!

18. Dinamikus memóriakezelés

Eddig, ha nagyobb mennyiségű adatot tároltunk, tömböket használtunk. Igen gyakran azonban fordítási időben nem lehet előre tudni, hogy mennyi adatot kell majd feldolgozni. Két eset van.

- Az egyik, ha azt tudjuk mondani, hogy „ennyi hely biztosan elég lesz”. Például sztringek beolvasásánál mást nem is nagyon tudunk csinálni, legfeljebb ha rendkívül megbonyolítjuk a programunkat.
- A másik eset, ha saccolni sem tudunk.

Mondhatnánk erre, hogy akkor foglaljunk pl. 100 millió elemű tömböt. Ez egyrészt nem gazdaságos, hiszen mi van akkor, ha az esetek túlnyomó többségében ennél sokkal kevesebb hely kell? Főlegesen foglalná a programunk a rendszer erőforrásait. Másrészt viszont mi van, ha nem elegendő a 100 millió sem? (Előre ismeretlen mennyiségű adatnak nagyméretű tömböt lefoglalni igen súlyos hiba.)



A megoldás, ha a memóriát nem előre kérjük, hanem a program futása közben, az aktuális igényeinknek megfelelően (dinamikusan) foglaljuk le, és ha már nincs rá szükség, felszabadítjuk. Az igényelt memóriát az operációs rendszer biztosítja a rendelkezésre álló szabad memória terhére. Ehhez a C nyelvben két memórafoglaló függvény áll rendelkezésre: a malloc és a calloc. A két függvény közötti különbségek a következők:

- A **malloc**-nak egy paramétere van: hány bajtot szeretnénk lefoglalni. Ezzel szemben a **calloc**-nak két paramétere van, és a kettő szorzata adja a kívánt bajtszámot.
- A malloc által lefoglalt memóriaterület „memóriaszemetet” tartalmaz, azaz a dinamikusan lefoglalt változó kezdeti értéke bármi lehet, ahogyan ezt az egyszerű, nem dinamikusan lefoglalt változóknál is megszoktuk. A calloc ezzel szemben a lefoglalt terület minden bajtját kinullázza.

Más különbség nincs, a használat és a felszabadítás ugyanaz mindkét esetben. A **függvények visszatérési értéke egy void* típusú pointer** a lefoglalt memóriaterületre, vagy ha nem sikerült lefoglalni (nincs a kívánt nagyságú összefüggő memóriaterület az operációs rendszer birtokában), akkor NULL pointer. A dinamikus memória kezeléséhez szükséges függvények használatához szerkesszük a programunkhoz az **stdlib.h** fejlécállományt!

A lefoglalt dinamikus memóriát kötelező felszabadítani a free függvénnyel!

Példa: a felhasználótól bekért méretű, double elemű dinamikus tömb létrehozása, elemeinek feltöltése az index értékével, kiírás fordított sorrendben, majd a memória felszabadítása.

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

int main(void){
    unsigned meret, i;
    double * t;
    printf("Mekkora tömb kell? ");
    scanf("%u", &meret);

    // memórafoglalás

    t=(double*)malloc(meret*sizeof(double));
    if(t==NULL){
        puts("Sikertelen allokálás.");
        exit(1);
    }

    // Innentől úgy használjuk, mint
    // egy közönséges tömböt.

    for(i=0; i<meret; i++)t[i]=i;
    for(i=meret-1; i+1!=0; i--)
        printf("%g ", t[i]);
```

```

// felszabadítás, ha már nem kell
free(t);
return 0;
}

```

A malloc helyett calloc-ot használva:

```
t=(double*) calloc(m, sizeof(double));
```

Figyeljük meg, hogy vessző került a csillag helyére!

A visszaadott `void*` pointert `double*` típusúvá konvertáljuk. Ez a konvertálás a valóságban nem csinál semmit, hiszen így is úgy is ugyanazt a memóriacímet jelenti. Azt a célt szolgálja, hogy tudassuk a fordítóval: valóban nem `void*`-ként, hanem `double*`-ként akarjuk használni a címet, tehát ne adjon hibaüzenetet (pontosabban warningot).

A lefoglalást és felszabadítást kivéve a lefoglalt változót ugyanúgy használhatjuk, mint egy nemdinamikus tömböt, illetve pointerrel mutatott változót, átadhatjuk pl. függvénynek is.

18.1 sizeof

Egy adattípus vagy változó méretét a **sizeof operátor** adja. (A többi operátortól eltérően ez szóveges operátor. A C-ben szóveges operátor még a típuskonverziós operátor: `(int)a`.) Azt mondja meg, hogy az adott adat hány bájtos (1 bájt nem biztos, hogy 8 bites). A `char` típus mindig 1 bájt, tehát a `sizeof(char)` mindig 1-et ad. A `sizeof` operátor fordítási időben kerül kiértékelésre, tehát a lefordított programba már a konkrét méretek lesznek behelyettesítve. Minden típus mérete egész bájtyszámú, tehát nincs pl. 1,5 bájtos változó.

```

double d;
int t[30];
int meret=sizeof(d); // változó méretét
// is nézhetjük
meret=sizeof(double); // típus méretét
// is nézhetjük
meret=sizeof(t); // hány bájtot
// foglal a tömb
meret=sizeof t; // hány bájtot
// foglal a tömb
meret=sizeof(t)/sizeof(int);
// hány elemű a tömb (int a tömb típusa)

```

Ha típusra alkalmazzuk a `sizeof`-ot, akkor a típust kerek zárójelek közé kell tenni, ha változóra ill. kifejezésre, akkor a kifejezést nem kell zárójelek közé tenni, de szabad. A `sizeof` nem használható inkomplet típusra, vagy definícióra (pl: `sizeof(int [])`).

Nagyon fontos! A sizeof nem tudja megállapítani a függvénynek átadott tömb méretét, mivel a függvény nem a tömböt, csak annak címét veszi át! Ezért adtuk át mindig a tömb méretét is a függvénynek. A `sizeof` kiértékelése mindig fordítási időben történik, azaz a lefordított programba már a konkrét értékek kerülnek be.



```

double summa(double [] t, unsigned n){
    int m=sizeof(t)/sizeof(double);
    double s=0.0;
    if(m==n)
        printf("Ez általában nem igaz!");
    for(i=0; i<n; i++) s+=t[i];
    return s;
}

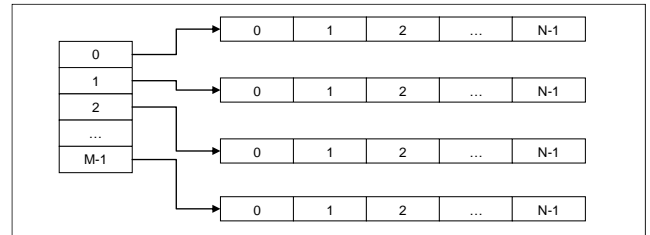
```

Itt tehát `m` a pointerméretet adja, `n` pedig a tényleges tömbméretet.

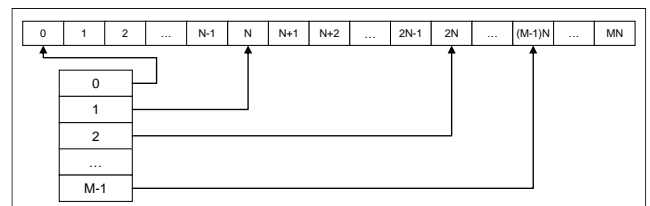
18.2 Többdimenziós dinamikus tömb

Dinamikus foglalással csak egydimenziós tömböt hozhatunk létre. Ha többdimenziósra van szükség, akkor több lehetőség áll előttünk. Pl. szükségünk van egy `M` soros, `N` oszlopos kétdimenziós tömbre:

- Lefoglalunk egy $M*N$ méretű egydimenziós tömböt, az i . sor j . oszlopában lévő elemet $t[i*N+j]$ módon érhetjük el.
- A $t[i][j]$ módon való címzéshez előbb egy M méretű pointertömböt foglalunk, majd M darab N méretű tömböt az elemeknek, és a pointertömb egy egy-eleme ezekre a sorokra mutat.



- Egy másik módszer a $t[i][j]$ módon való címzéshez: először szintén egy M méretű pointertömböt foglalunk, majd egy darab $M*N$ méretű tömböt az elemeknek. A pointertömb egy-egy eleme ezen a tömbön belül N elemenként eltolva mutat az elemekre. Ez a megoldás hasonlít legjobban a nemdinamikus többdimenziós tömbök működésére.



Példa ▶ Kérjük a felhasználótól a kétdimenziós tömb méretét, hozzuk létre, töltsük fel adatokkal, majd írjuk ki, végül töröljük!

1. változat:

```

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

void hibaelloporzo(void * p){
    if(p==NULL){
        fprintf(stderr, "Sikertelen allokacio.\n");
        exit(1);
    }
}

int main(void){
    int M,N,i,j;
    double ** tomb;

    // Méretek bekérése

    printf("M=");
    scanf("%d", &M);
    printf("N=");
    scanf("%d", &N);

    // pointertömb foglalása

    tomb=(double**) calloc(M, sizeof(double*));
    hibaelloporzo(tomb);

    // sorok foglalása

    for(i=0; i<M; i++){
        tomb[i]=(double*) calloc(N, sizeof(double));
        hibaelloporzo(tomb[i]);
    }
}

```

```

// használat
for(i=0; i<M; i++)
    for(j=0; j<N; j++)
        tomb[i][j]=0.1*i*j;

for(i=0; i<M; i++){
    for(j=0; j<N; j++)
        printf("%9g ",tomb[i][j]);
    printf("\n");
}

// sorok felszabadítása
for(i=0; i<M; i++)free(tomb[i]);

// pointertömb felszabadítása
free(tomb);

return 0;
}

```

2. változat:

```

int main(void){
    int M,N,i,j;
    double ** tomb;

    // Méretek bekérése
    printf("M=");
    scanf("%d",&M);
    printf("N=");
    scanf("%d",&N);

    // pointertömb foglalása
    tomb=(double**)calloc(M,sizeof(double*));
    hibaellelnozo(tomb);

    // sorok foglalása
    ! tomb[0]=(double*)calloc(M*N,sizeof(double));
    ! hibaellelnozo(tomb[0]);
    ! for(i=0; i<M; i++)
    !     tomb[i]=tomb[0]+N*i;

    // használat
    for(i=0; i<M; i++)
        for(j=0; j<N; j++)
            tomb[i][j]=0.1*i*j;

    for(i=0; i<M; i++){
        for(j=0; j<N; j++)
            printf("%9g ",tomb[i][j]);
        printf("\n");
    }

    // sorok felszabadítása
    ! free(tomb[0]);

    // pointertömb felszabadítása
    free(tomb);

    return 0;
}

```

A két változat a két ábrának megfelelő, különbség a felkiáltójellel jelzett sorokban van.

A *hibaellelnozo* függvény csak azt vizsgálja, hogy a pointer NULL-e, bármilyen pointertípust használhatunk. Mivel többféle típussal is meghívjuk, a legcélszerűbb típus nélküli pointert megadni paraméterként.

18.3 A lefoglalt memória bővítése

A bemutatott példában megkérdeztük a felhasználót, mekkora tömbre lesz szüksége, és ekkorát foglaltunk. Gyakori azonban, hogy egyáltalán nem tudjuk előre, hogy mennyi elemre lesz

szükség. Az eddig megismert eszközök segítségével kell megoldanunk a problémát.

Meg lehet-e nyújtani a lefoglalt memóriát? Nem.

Ha nem elegendő a lefoglalt dinamikus tömb, a következő módszerrel tudjuk „magnövelni a méretét”:

- *t* pointer mutat a dinamikus tömbre, melynek *M* a mérete. *N* méretű helyre lenne szükségünk.
- Lefoglalunk egy *N* méretű helyet, melyre *v* pointerrel mutatunk.
- Átmásoljuk az elemeket a *t* tömbből a *v* tömbbe.
- Felszabadítjuk *t* tömböt, és *t* pointert *v* tömbre állítjuk.

```

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

```

```

// ha kisebb az új tömb, akkor csak ennyi
// elemet másolunk át a régiből
double*ujrafoglal(double *t,int M,int N){
    double * v;
    unsigned i, min = M<N ? M : N;

```

```

    v=(double*)calloc(N,sizeof(double));
    if(v==NULL)return NULL;
    for(i=0;i<min;i++)v[i]=t[i];
    free(t);
    return v;
}

```

```

void hiba(char * s){
    fprintf(stderr,"%s\n",s);
    exit(1);
}

```

```

int main(void){
    double * t;

    t=(double*)malloc(100*sizeof(double));
    if(t==NULL)hiba("Memoriafoglalas");

    // magnöveljük 200 eleműre

    t=ujrafoglal(t,100,200);
    if(t==NULL)hiba("Memoriafoglalas");

    // lecsökkentjük 50 eleműre

    t=ujrafoglal(t,200,50);
    if(t==NULL)hiba("Memoriafoglalas");

    free(t);
    return 0;
}

```

Létezik egy C függvény, a *realloc*, amely ugyanezt teszi. A fenti program *realloc*-os változata:

```

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

```

```

void hiba(char * s){
    fprintf(stderr,"%s\n",s);
    exit(1);
}

```

```

int main(void){
    double * t;
    t=(double*)malloc(100*sizeof(double));

```

```

if (t==NULL) hiba ("Memoriafoglalás");

// megnöveljük 200 eleműre

t=(double*) realloc(t,200);
if (t==NULL) hiba ("Memoriafoglalás");

// lecsökkentjük 50 eleműre

t=(double*) realloc(t,50);
if (t==NULL) hiba ("Memoriafoglalás");

free (t);
return 0;
}

```

A módszer hátrányai:

- ideiglenesen többletmemóriát igényel
- hosszadalmas az eredeti tömb átmásolása

Ezek a hátrányok kiküszöbölhetők **önhivatkozó adatszerkezetek** alkalmazásával, melyekről a 24. és 27. fejezetben lesz szó.

Gondolkozzunk együtt!

Oldja meg egyedül!

19. Sztringek

A tömbökkel foglalkozó fejezetben megismerkedtünk a sztringek felépítésével (karakter sorozat egy tömbben, melynek végét a nulla értékű karakter jelzi). Most megnézzük a könyvtári sztringkezelő függvények közül a fontosabbakat és néhány sztringalgoritmust.

19.1 Könyvtári sztringfüggvények

A sztringekkel kapcsolatosan leggyakrabban szükséges feladatok és az ezeket végrehajtó könyvtári függvények a következők:

- hossz megállapítása: *strlen*
- másolás: *sprintf*, *strcpy*
- összefűzés: *sprintf*, *strcat*
- összehasonlítás: *strcmp*
- sztring számmá alakítása: *sscanf*, *atoi*, *atof*, *atol*
- szám sztringgé alakítása: *sprintf*
- keresés sztringen belül: *strstr*, *strchr*

sprintf, **sscanf** ► A jól ismert printf és scanf függvények stringbe dolgozó változatai. A sprintf a képernyő helyett az első paraméterként kapott karaktertömbbe írja a szöveget, a sscanf pedig a billentyűzet helyett az első paraméterként kapott sztringből olvassa ki az értékeket.

A következő példaprogram igyekszik felvillantani a két függvény lehetőségeit.

```

#include <stdio.h>

void kiir(int a,int b,char*t,char*s){
    printf("t=%s\ na=%d\ nb=%d\ ns=%s\n\n",t,a,b,s);
}

int main(void){
    char t[]="65 ember osszjovedelme 877323 Ft.";
    char s[40];
    int a,b,i;

    sscanf(t,"%d%s%s%d",&a,s,&b);
    kiir(a,b,t,s);

    sscanf(t+25,"%d%s",&b,s);
    kiir(a,b,t,s);

    sprintf(t+4,"%s+%d szia!",s,a+b+3);
    kiir(a,b,t,s);

    printf("A t tomb tartalma:\n");
    for(i=0;i<sizeof(t);i++)
        printf("t[%2d] = %3d = '%c'\n",
            i,t[i],t[i]);
    return 0;
}

```

A kód áttekinthetőségének javítása érdekében a négy változó képernyőre való kiírását külön függvénybe tettük.

A *sscanf(t,"%d%s%s%d",&a,s,&b)*; a *t* tömbből olvas három értéket: az *a* változóba kerül a 65, az *s* változóba az „ember”, a *%s*-ben a *** azt jelzi, hogy a következő szót átugorjuk, nem olvassuk be sehová, így a *b* változóba a 877323 kerül. Az első kiírás eredménye tehát:

```

t=65 ember osszjovedelme 877323 Ft.
a=65
b=877323
s=ember

```

A *sscanf(t+25,"%d%s",&b,s)*; utasításban bemeneti sztringként *t+25* szerepel, vagyis a *t* pointerhez 25-öt adunk, ami a *t* tömb 25. elemének címét jelenti. A *sscanf* függvény számára ez a „7323 Ft.” tartalmú tömböt jelenti bemenetként, ebből fog olvas-

ni. Egy egész számot és egy sztringet olvasunk tehát, így a kiírás eredménye:

```
t=65 ember osszjovedelme 877323 Ft.
a=65
b=7323
s=Ft.
```

A `sprintf(t+4, "%s+%d szia!", s, a+b+3);` utasítással lecseréljük az eredeti sztringet a `t` tömbben. Bemenete a `t+4`, azaz a `t` tömb elején lévő „65 e” részt nem bántjuk, utána kerül az új kiírás. Az eredmény:

```
t=65 eFt.+7391 szia!
a=65
b=7323
s=Ft.
```

Érdekes megnézni a `t` tömb teljes tartalmát is, amit egy ciklussal írunk ki, előbb a karakter ASCII kódját, aztán magát a karaktert aposztrófok között.

```
A t tömb tartalma:
t[ 0] = 54 = '6'
t[ 1] = 53 = '5'
t[ 2] = 32 = ' '
t[ 3] = 101 = 'e'
t[ 4] = 70 = 'F'
t[ 5] = 116 = 't'
t[ 6] = 46 = '.'
t[ 7] = 43 = '+'
t[ 8] = 55 = '7'
t[ 9] = 51 = '3'
t[10] = 57 = '9'
t[11] = 49 = '1'
t[12] = 32 = ' '
t[13] = 115 = 's'
t[14] = 122 = 'z'
t[15] = 105 = 'i'
t[16] = 97 = 'a'
t[17] = 33 = '!'
t[18] = 0 = ' '
t[19] = 108 = 'l'
t[20] = 109 = 'm'
t[21] = 101 = 'e'
t[22] = 32 = ' '
t[23] = 56 = '8'
t[24] = 55 = '7'
t[25] = 55 = '7'
t[26] = 51 = '3'
t[27] = 50 = '2'
t[28] = 51 = '3'
t[29] = 32 = ' '
t[30] = 70 = 'F'
t[31] = 116 = 't'
t[32] = 46 = '.'
t[33] = 0 = ' '

```

Figyeljük meg, hogy a „65 eFt.+7391 szia!” szöveget a lezáró nulla követi, majd az eredeti sztring megmaradt része.

A `sizeof(t)` a tömb méretét adja bajtban. Ha egy `int` tömböt akarnánk végigjárni, a ciklus így nézne ki:

```
int egesz[]={6,2,77,21,4},i;
for(i=0; i < sizeof(egesz)/sizeof(int); i++)
    printf("egesz[%d] = %d\n",i,egesz[i]);
```

A `sizeof(egesz)` ugyanis a tömb méretét adja bajtokban, nekünk viszont az elemszámra van szükségünk, amit egy elem méretével való



osztással kapunk. A karaktertömbnél az osztásra azért nem volt szükség, mert a `char` típus mérete definíció szerint 1 bajt, azaz `sizeof(char)==1`. Természetesen helyes lenne a program akkor is, ha így írnánk: `i<sizeof(t)/sizeof(char)`.

strlen, strcpy, strcat, strcmp, strchr, strstr, strncpy ► Egy példaprogramon keresztül mutatjuk be ezek használatát:

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

int main(){
    char t[100];
    char t1[]="Egy meg egy";
    char t2[]=" az ketto.";
    printf("%d\n",strlen(t1)); // NEM számolja a
                                // lezáró 0-t!

    strcpy(t,t1);
    puts(t);
    strcat(t,t2);
    puts(t);
    strcpy(t1,"Harry");
    strcpy(t2,"Ron");
    strcpy(t,"Hermione");
    if(strcmp(t1,t2)<0)printf("%s<%s\n",t1,t2);
    else printf("%s>=%s\n",t1,t2);
    if(strcmp(t1,t)>0)printf("%s>%s\n",t1,t);
    else printf("%s<=%s\n",t1,t);
    if(strcmp(t1,t)==0)printf("%s==%s\n",t2,t);
    else printf("%s!=%s\n",t2,t);
    if(strchr(t,'o')==NULL)
        printf("%s nem tartalmaz 'o'-'t\n",t);
    if(strstr(t,"on")==NULL)
        printf("%s nem tartalmaz 'on'-'t\n",t);
    else printf("%s tartalmaz 'on'-'t: %s\n",t,
        strstr(t,"on"));
    strncpy(t1,"Beckham",2);
    puts(t1);
    return 0;
}
```

A képernyőn ez jelenik meg:

```
11
Egy meg egy
Egy meg egy az ketto.
Harry<Ron
Harry<=Hermione
Ron!=Hermione
Hermione tartalmaz "on"-t: one
Berry
```

- `strlen(t1)`: a `t1` sztring hosszát adja, a lezáró 0-t nem számolja bele
- `strcpy(t,t1)`: átmásolja `t`-be `t1`-et, ugyanaz, mint a `sprintf(t, "%s", t1);`, bármelyik megoldást használhatjuk. (Az értékadás iránya itt is jobbról balra, ahogy az = operátornál.)
- `strcat(t,t2)`: `t` végéhez fűzi `t1`-et. Az összefűzés a `sprintf` segítségével is megoldható, de a forrás és a cél tömb nem lehet ugyanaz, azaz a `sprintf(t, "%s%s", t, t2);` hibás! A fenti programban a kívánt eredményt így kapjuk `sprintf` használatával: `sprintf(t, "%s%s", t1, t2);`.



Magyarázat: a `sprintf` függvény a tömbök címét veszi át, és nem ellenőrzi, hogy van-e átfedés a tömbök között, egyszerűen elkezdje beleírni a cél tömbbe azt, amire a formátumsztringje utasítja. Ha a forrás és a cél ugyanaz, akkor azzal, hogy a célba beleír, a forrást is módosítja, ami problémás lehet, például ha felülírja a forrás lezáró nulláját egyéb karakterrel.

- `strcmp(t1,t2)`: összehasonlítja `t1`-et és `t2`-t.
 - Ha egyformák, 0 a visszatérési értéke.
 - Ha `t1 < t2`, azaz ABC-ben előbb van, akkor a függvény negatív értéket ad vissza (nem definiált, hogy mit, a lényeg, hogy negatív).
 - Ha `t1 > t2`, akkor pozitív.

- `strchr(t,ch)`; megnézi, hogy *t* tartalmazza-e a *ch* karaktert. Ha nem, akkor NULL pointert ad vissza. Ha tartalmazza, akkor az első megtalált karakter címét a *t*-n belül.
- `strstr(t1,t2)`: megnézi, hogy *t1* tartalmazza-e a *t2* szöveget. Ha nem, akkor NULL pointert ad vissza. Ha tartalmazza, akkor az első megtalált részsstring címét a *t1*-en belül.
- `strncpy(t1,t2,n)`: *t2*-ből *n* db karaktert másol *t1*-be. Ha *n* kisebb, mint a string hossza, akkor nem másolja át a lezáró 0-t, ennek a következménye, amit a példaprogram mutat: az első két karakter felülíródik a „Beckham” első két betűjével, a „Harry” többi része ott marad, a kiíró függvény (jelen esetben a `puts`, de a `printf` is ugyanígy viselkedne) pedig mindig a lezáró 0-ig írja ki a szöveget. Ha *n* nagyobb, mint a másolandó szöveg hossza, akkor a szöveg vége után nulla bájtokkal töltődik fel a tömb *n*-ig.

További függvények állnak rendelkezésre a `string.h`-ban, sok függvénynek van *n*-es változata az `strncpy` mintájára (pl. `strncmp` az első *n* karaktert hasonlítja össze). Amit célszerű megjegyezni, az az `strlen`, `strcpy` és `strcat`. A többit szükség esetén megtaláljuk a Helpben.

Sztring-szám és szám-sztring átalakítások ► Láttuk már az átalakítást a `scanf` és az `sprintf` segítségével. A sztringből számmá alakításhoz rendelkezésre áll néhány speciális függvény is, az `atoi`, `atof`, `atol`. mindhárom bemenete egy sztring, visszatérési értékük pedig `int`, `double` illetve `long`. Használatukhoz az `stdlib.h` beszerkesztése szükséges. Pl.:

```
char t[]="65 ember osszjovedelme 877323 Ft.";
int a;
a=atoi(t);
```

Az *a*-ba 65 kerül. Ha a bemeneti sztring számként nem értelmezhető, a függvények nullát adnak vissza. Ezt az esetet nem lehet megkülönböztetni attól, ha tényleg nulla van a bemenő sztringben, emiatt a hibakezelés problémás. A függvények az első olyan karakterig olvasnak, ami nem lehet az adott típusú szám része, pl. a „-45.21x” az `atoi` és az `atol` számára -45, az `atof` számára -45.21 (a `scanf`, `scanf` stb is ugyanígy működik).

Hibakezelésre alkalmasabb a `sscanf`:

```
if(sscanf(t+25,"%d%s",&b,s)!=2){
    fprintf(stderr,"Hibas bemeno adat.\n");
    exit(1);
}
```

Két értéket szeretnénk beolvasni, ha ez nem sikerül, akkor baj van.

19.2 Algoritmusok

Írjuk meg magunk a könyvtári sztringkezelő függvényeket! Nem mindet persze, de néhányat.

strlen ►

```
size_t x_strlen(const char *s){
    size_t i;
    for(i=0; s[i]!=0; i++);
    return i;
}
```

Az `x_strlen` ugyanazt teszi, mint az `strlen`. Visszatérési típusa `size_t`, ami egy `typedef` segítségével definiált szabványos típus, sok szabványos függvény használja méretmegadásra. Használhatunk helyette `int`-et vagy `unsigned`-et, ha az jobban tetszik.

A paraméter típusában szereplő `const` azt jelenti, hogy a függvény nem változtatja meg az *s* tömb tartalmát, ezt következ-

mény nélkül el lehet hagyni. Ha a függvény írója véletlenül megváltoztatná a sztringet, a `const` miatt a fordító hibáüzenetet adna. A függvény használatát semmiben nem befolyásolja.

strcpy ► A következő algoritmusunk a sztringmásolás. Ennek négy változatát is megadjuk:

```
char * x_strcpy_1(char * ki,const char * be){
    int i;
    for(i=0; be[i]!=0; i++)ki[i]=be[i];
    ki[i]=0; // lezáró!
    return ki;
}

char * x_strcpy_2(char * ki,const char * be){
    int i;
    for(i=0; (ki[i]=be[i])!=0; i++);
    return ki;
}

char * x_strcpy_3(char * ki,const char * be){
    int i;
    for(i=0; ki[i]=be[i]; i++);
    return ki;
}

char * x_strcpy_4(char * ki,const char * be){
    while(*ki++=*be++);
    return ki;
}
```

Az elsőtől a negyedikig haladva egyre több C trükköt használunk. Elég, ha az első változatot meg tudjuk írni, a többit csak érdekesség kedvéért közöltük.

Az elsőben a lezáró nullát nem másoljuk át, mert véget ér a ciklus, ezért ezt oda kell biggyeszteni a sztring végére. A többiben a másolás a ciklus feltételében van, és csak a lezáró nulla átmásolását követően nézzük meg, hogy mit is másoltunk.

A negyedik változat a *be* pointer által mutatott értéket a *ki* pointer által mutatott címre másolja, ezt követően mindkét pointer értékét megnöveli eggyel.

strcat ► Sztringek összefűzése:

```
char * x_strcat_1(char * beki,const char * be2){
    int i,j;
    for(i=0; beki[i]!=0; i++);
    for(j=0; (beki[i]=be2[j])!=0; i++,j++);
    return beki;
}

char * x_strcat_2(char * beki,const char * be2){
    strcpy(beki+strlen(beki),be2);
    return beki;
}

char * x_strcat_3(char * beki,const char * be2){
    char *p=beki;
    while(*p++);
    p--; // vissza a lezáró 0-ra
    while(*p++=*be2++);
    return beki;
}
```

Figyeljük meg az első változatban, hogy két külön indexet használtunk a két sztringhez! A 2-3 függvényváltozat ezúttal is csak érdekesség.

strcmp ►

```
int x_strcmp(const char *a,const char *b){
    int i;
    for(i=0; a[i]!=0 && b[i]!=0 && a[i]==b[i]; i++);
    if(a[i]<b[i])return -1;
    if(a[i]>b[i])return 1;
    return 0;
}
```

A ciklus addig megy, míg valamelyik sztring véget nem ér, vagy amíg a két sztring egyforma.

Ha a két sztring egyforma, akkor $a[i]$ és $b[i]$ egyaránt a lezáró nullán fog állni, azaz ekkor sem $a[i] < b[i]$, sem $a[i] > b[i]$ nem teljesül, tehát a függvény 0-t ad vissza.

Ha a két sztring különbözik, akkor az első különböző karakter alapján döntjük el, hogy melyik van előbb az ABC-ben.

Ha az összehasonlítás úgy ér véget, hogy az egyik sztring a másik eleje, vagyis pl. $a = \text{"Konyha"}$, $b = \text{"Konyhaablak"}$, akkor a rövidebb sztring $[i]$ eleme a lezáró nulla, míg a másik sztring esetében egy nem 0 érték van ott (a példában 'a'), ezt hasonlítjuk össze.

Mi van akkor, ha az egyik sztring a másik eleje, és a hosszabbik egy ékezetes magyar magánhangzóval folytatódik? Ugyanúgy rendez-e a szövegeket, ha a char előjeles illetve előjel nélküli az adott rendszerben?

Összefűtés ▶ Végül nézzünk egy olyan példát, amire nincs könyvtári függvény:

Írjunk függvényt, amely paraméterként kap két sztringet és ezeket összefűteli úgy, hogy az eredménybe felváltva kerüljenek be a karakterek a két forrásból, amíg lehet, aztán a hosszabbik sztring maradék karakterei következzenek egymás után! Az eredmény karaktértömböt szintén paraméterként kapja a függvény. Pl.: be: „0123456789” és „abcdef”, ki: „0a1b2c3d4e5f6789”.

Mielőtt megnézi a megoldást, próbálja meg egyedül megoldani a feladatot!

A két sztring eltérő hossza miatt nem lehet egy indexszel megoldani a feladatot. Sokféle helyes algoritmus létezik, az alábbi ezek közül csak egy.

```
void fesul(char ki[], char be1[], char be2[]) {
    int i=0, j=0, k=0;
    while (be1[i] != 0 || be2[j] != 0) {
        if (be1[i] != 0) ki[k++] = be1[i++];
        if (be2[j] != 0) ki[k++] = be2[j++];
    }
    ki[k] = 0; // lezáró
}
```

A ciklus akkor ér véget, ha mindkét bemenő sztring végére értünk. i és j értékét csak akkor növeljük, ha még nem vagyunk a $be1$ ill. a $be2$ végén. A $ki[k++] = be1[i++]$; szűtszedve így írható: $\{ ki[k] = be1[i]; k++; i++; \}$, mivel a ++ mindkét esetben posztinkrement, azaz a kifejezés kiértékelése után növelik a változó értékét.

Írjon függvényt, amely egy sztringet kap paraméterként, melyet saját helyén megfordít (azaz nem használ másik tömböt vagy egyéb adatszerkezetet a megfordított vagy az eredeti sztring ideiglenes tárolására)!

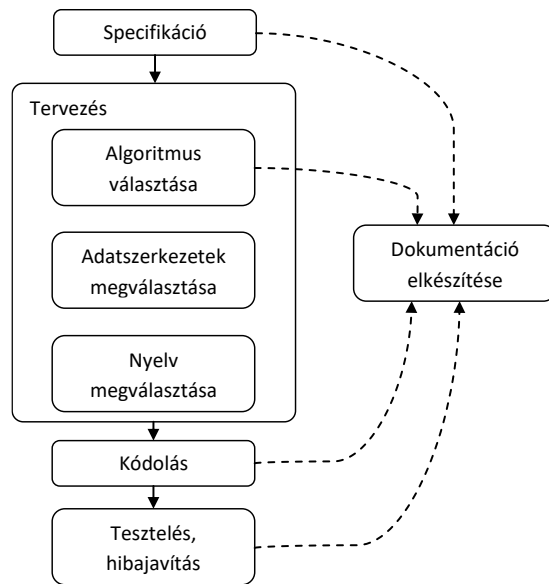
Gondolkozzunk együtt!

Oldja meg egyedül!

20. A programozás menete

Nagyobb programok készítésekor nem célszerű úgy eljárni, hogy az ember leül a gép elé és elkezd kódolni. A program felépítését meg kell tervezni, a készítés folyamatát pedig dokumentálni kell, hogy a későbbiekben könnyű legyen karbantartani illetve fejleszteni a szoftvert. Az is gyakori, hogy több programozó dolgozik ugyanazon szoftver különböző részein. Beláthatjuk, hogy a program megtervezése és a feladatok előzetes felosztása nélkül az együttműködés lehetetlen lenne. Ebben és a következő fejezetben a nagyobb programok készítéséhez kapcsolódó dolgokról lesz szó.

Emlékeztetőül a programozás menetét bemutató ábránk az 1. fejezetből, illetve az egyes lépések bővített leírása:



- **Specifikáció:** a program feladatának részletes definiálása, beleértve a bemeneti és kimeneti adatok formátumát.
- **Algoritmus választása:** a program felépítésének megtervezése a program megírásához szükséges részletességgel. Fontos szempont a hatékonyság (műveletvégzés, elvi korlátok, memóriaigény futás közben), a program mérete, a programfejlesztés sebessége.
- **Adatszerkezet megválasztása:** a feladathoz leginkább megfelelő struktúra megtervezése (pl. tömb, lista, fa?)
- **Nyelv megválasztása:** Mindig a feladatnak leginkább megfelelő nyelvet válasszuk. Pl. website fejlesztéshez alkalmasabb a php, mint a C.
- **Kódolás, tesztelés, hibajavítás:** ezek a lépések egymást váltogatják. Nagy programoknál jellemző, hogy hibakereséssel több idő megy el, mint a programozás összes többi tevékenységével együttvéve.
- **Dokumentáció elkészítése:**
 - **Programozói (fejlesztői) dokumentáció:** fontos, hogy később javítani, bővíteni lehessen a programot, és erre ne csak a program eredeti írója legyen képes, hanem más programozó is.

A fejlesztői dokumentáció legfontosabb eszköze maga a jól kommentezett, világos, áttekinthető struktúrájú forráskód. Rengeteg időt lehet megtakarítani, ha nem kell elmélyedni a forráskódban, és megpróbálni kitalálni, hogy „mit is jelentett a c változó?”, mert a kommentekre pillantva másodpercek alatt behatárolható a keresett programrész, és beszédes változó- és függvénynevek alapján ezen belül is könnyű tájékozódni. (Az egy-két betűs változónév is lehet beszédes.

i, j, k a ciklus futóváltozója, n a darabszám, x, y koordináták, fp az egy szem megnyitott fájl, ezek teljesen egyértelműek, ha konzervensen használjuk az elnevezéseket.)

A dokumentáció készítéséhez nagy segítséget nyújt a Doxygen program, mely a C kódban elhelyezett speciális felépítésű megjegyzések segítségével készít dokumentációt.

A programozói dokumentáció része a forráskód mellett a specifikáció, adatszerkezetek (beleértve a fájlok formátumát), és az algoritmusok leírása, diagramok. Mindaz, ami ahhoz szükséges, hogy valaki gyorsan áttekinthetést szerezzen a programról, és aztán könnyen eligazodjon a részletekben.

- **Tesztelési dokumentáció:** ha elkészült a program, vagy annak valamely modulja, akkor azt tesztelni kell, hogy felderítsük az esetleges hibákat. A tesztelési dokumentációban fel kell tüntetni, hogy milyen tesztadatokkal vizsgáltuk a szoftvert, milyen hibákat találtunk, és javítottunk. Későbbi hibakeresések során hasznos lesz akár kiinduló adatnak, akár azért, hogy segítsen, mit nem kell már tesztelni.
- **Felhasználói dokumentáció:** magyarul használati utasítás. A program bonyolultságától függ ennek összetettsége. Tartalmazza a szoftver funkcióit, a be- és kimeneti adatok, fájlok formátumait, melyekkel a felhasználó a programmal kommunikál, a korlátokat, ismert hibák jegyzékét, a szoftver által támasztott hardver és szoftver követelményeket, a fejlesztő elérhetőségét.

20.1 Programstruktúra tervezése

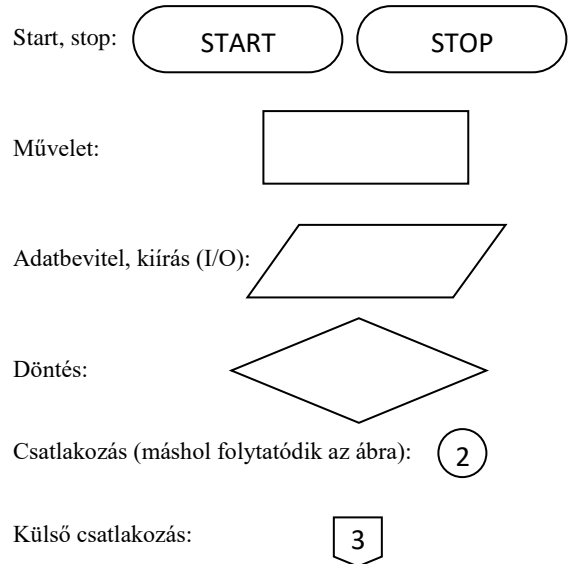
Egy nagyméretű, sok ezer soros programot nem kezdhetünk el úgy írni, hogy van egy homályos elképzelés a fejünkben arról, hogy is fog felépülni a program, mert ennek szinte biztosan az lesz a következménye, hogy oda jutunk: a legjobb lenne előről kezdeni az egészet, annyira kusza és nehézkes, amit alkottunk. Meg kell tervezni a programot: tervezés közben hamar kiderülnek az előzetes elképzelésünkben rejlő legnagyobb strukturális hibák, melyek ilyenkor még egyszerűen javíthatók. A tervezés során kétféle módszert használunk:

- **top-down (felülről lefelé) tervezés:** A specifikáció alapján meghatározzuk a program fő részeit (pl. felhasználói felület, 3D grafikus motor, mesterséges intelligencia, hangkezelés, hálózatkezelés, stb.), tisztázzuk, hogy melyik rész mit csinál (=specifikációt készítünk a részekhez), majd az egyes részeket bontjuk kisebb részekre, a kisebb részeket specifikáljuk, és így tovább. Addig bontjuk a feladatot, amíg azt nem tudjuk mondani: „Ezen a blokkon belül minden feladat világos, szükségtelen tovább bontani.”
- **bottom-up (alulról felfelé) tervezés:** Ha egy adott blokkban világos a feladat, tehát szétbontással nem jutunk már sehová, akkor nekiállunk elemi részekből összerakni a blokkot. Az elemi részek lehetnek utasítások (ciklus, feltételes elágazás stb.), függvények, objektumok, más által fejlesztett blokkok. A bottom-up tervezés nem is mindig válik el a kódolástól, hiszen a kódolás során általában ugyanazok az alapegységek, mint a bottom-up tervezésnél.

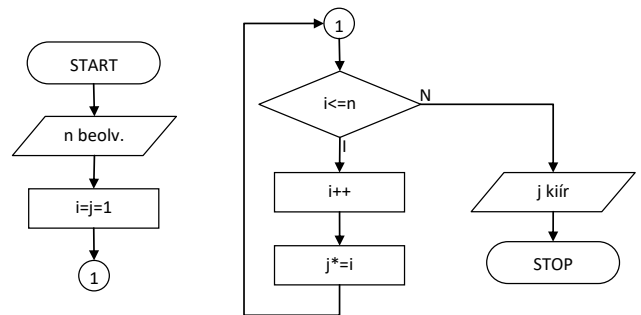
20.2 Algoritmus megadása

Két algoritmus megadási móddal ismerkedünk meg részletesebben: a folyamatábrával és a struktogrammal.

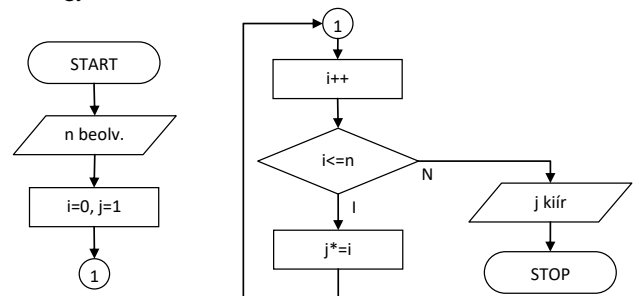
Folyamatábra ► A szöveges leírásnál könnyebben áttekinthető a grafikus algoritmusábrázolás. Ennek egyik legjobb eszköze a folyamatábra. A legegyszerűbb folyamatábrák mindössze két elemet használnak: a műveletet és a döntést, melyek segítségével minden algoritmus leírható. A gyakorlatban ennél bővebb elemkészlettel is dolgoznak az egyes funkciók jobb elkülönítése érdekében. Az alábbi lista további elemekkel bővíthető, számunkra azonban ez is bőven elegendő.



Példa: $n!$ kiszámítása szorzással.

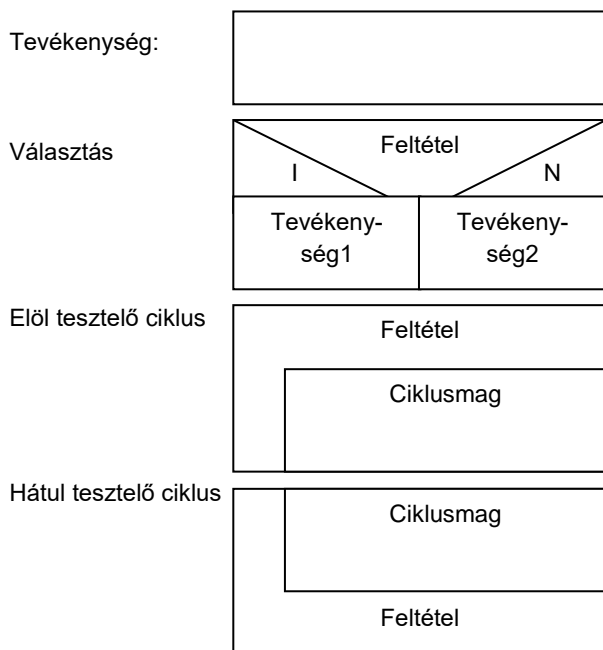


Folyamatábra segítségével könnyű jól áttekinthető algoritmust rajzolni, azonban miután megrajzoltuk az algoritmust, könnyen azt találhatjuk, hogy strukturált módon nem valósítható meg, mert olyan ugrások vannak benne, amit a strukturált utasítások (feltételes elágazás, ciklus) nem támogat. Például, ha a fenti ábrát így alakítanánk át:

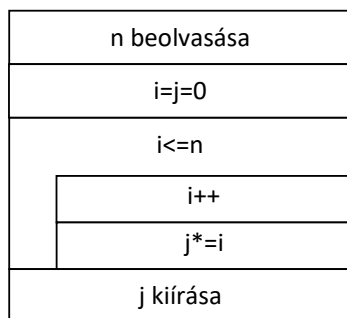


Ez az algoritmus működőképes, azonban hogy valósítanánk meg ezt a tanult C eszközökkel strukturált módon? Az algoritmus megváltoztatása nélkül sehog. A goto utasítás segítségével persze megvalósítható, de annak nem megengedett a használata pont azért, mert strukturálatlan, és követhetlenné teszi a kódot.

Struktogram ► A struktogram (struktúradiagram) szintén az algoritmusok grafikus ábrázolására való, de csak strukturált megoldásokat enged meg. Kevésbé szemléletes, mint a folyamat-ábra, azonban hasznosabb, ha le is akarjuk kódolni az algoritmust. A struktogram elemei egymáshoz csatlakozó téglalapok.



Példa: n! kiszámítása szorzással.



A struktogram téglalapjai nem feltétlenül egyforma szélességűek, pl. a ciklus magjában lehetne feltételes elágazás, annak az igaz ágán egy másik ciklus, stb.

21. Összetettebb C programok kellékei

21.1 Több modulból álló programok

Nagyobb programoknál célszerű nem egy forrásfájl használni, hanem több fájlra bontani a programot. Ez számos előnnyel jár:

- A fordító forrásfájlanként külön-külön fordít. Ha csak egy forrásfájlt módosítunk, akkor csak ezt kell ismét lefordítani.
- A nagy programokat gyakran több fejlesztő készíti, mind-egyikük a saját forrásfájljain dolgozik.
- Egy adott funkcióhoz tartozó programrészek kerülnek egy fájlba, így hamarabb megtalálható a keresett kódrészlet.

A C nyelvben egy függvény csak akkor használható, ha a deklarációját a fordítóprogram már megtalálta az adott forrásállomány lefordításakor. Ha külön modulba tesszük a függvények egy részének definícióját (azaz forráskódját), akkor szükség van arra, hogy a másik modul rendelkezésére bocsássuk a deklarációkat. Erre a célra használjuk a fejléc állományokat.

Példa ► Írjunk C függvényt, mely paraméterként két előjel nélküli egészt vesz át, és visszaadja a két szám legnagyobb közös osztóját! Egészítsük ki teljes programmá, amely teszteli a függvény helyes működését! A függvényt tegyük külön modulba!

Inko.h:

```
#ifndef LNKO_H
#define LNKO_H

unsigned lnko(unsigned a, unsigned b);

#endif
```

A fejlécfájl a deklaráción kívül a preproceszornak szóló feltételes fordítási utasításokat is tartalmaz. Ezek azt a célt szolgálják, hogy a fejlécfájl tartalmát a fordító csak egyszer fordítsa bele egy adott forrásfájlba. Normál esetben egyszer szerepel egy fejlécállomány beszerkesztése egy fájlban, azonban gyakran előfordul, hogy az egyik fejlécállományt beszerkeszti egy másik, és a forrásfájlban erre nem figyelve mindkettőt beszerkesztjük. Ami viszont valódi problémát okozhat: ha az egyik fejlécbe beszerkesztjük a másikat, a másikba pedig az egyiket. Ha ez ellen nem védekezünk a fenti módon, akkor a fordító nem tudja lefordítani a kódot, mert ez egy végtelen rekurzió.

Ha a `#ifndef` után megadott szimbólum (`LNKO_H`) már definiálva van, akkor a fordító nem szerkeszti be a `#endif`-ig tartó részt, ha még nincs definiálva, akkor beszerkeszti. A következő sorban definiáljuk (a `#define`-nal definiált konstansok és makrók esetében is ugyanígy vizsgálható a szimbólum létezése.)

Inko.c:

```
#include "lnko.h"

unsigned lnko(unsigned a, unsigned b){
    unsigned i=a<b?a:b;
    for(;i>1;i--){
        if(a%i==0 && b%i==0) return i;
    }
    return 1;
}
```

A fejlécfájl nevét "" között adtuk meg, mert a programunk mappájában található, nem pedig a rendszer mappában. Itt jelen esetben ez a beszerkesztés el is hagyható, azonban célszerű így írunk a programjainkat, mert ha a fejlécben deklaráljuk az összes függvényt, akkor a forrásállományban nem kell arra figyel-nünk, hogy az egyik függvényben használt függvény megelőzze a másikat.

main.c:

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

int main(void) {
    printf("lnko(273,865)=%u\n"
        ,lnko(273,865));
    printf("lnko(342,88395)=%u\n"
        ,lnko(342,88395));
    printf("lnko(97842834,762321)=%u\n"
        ,lnko(97842834,762321));
    printf("lnko(8192,6144)=%u\n"
        ,lnko(8192,6144));
    printf("lnko(836525,43210)=%u\n"
        ,lnko(836525,43210));
    return 0;
}
```

Úgy használjuk az lnko-t, mint a könyvtári függvényeket.

A C fordítóval futtatható programokon kívül létrehozhatunk függvénykönyvtárakat is. Visual C-ben ehhez Library Projectet kell indítani. A könyvtár projekt ugyanolyan .c és .h fájlokból áll, mint a futtatható programok projektjei, de nincs *main* függvény. A fordítás után létrejövő .lib fájlt hozzá kell adni az azt használó program linker beállításainak megfelelő sorához. (A szabványos függvényeket (printf, fopen, sqrt stb.) ugyanilyen lib fájlok tartalmazzák.) A függvények használatához szükség van a deklarációkat tartalmazó fejlécállomány (elkészítésére és) beszerkesztésére.

21.2 A C programok általános felépítése

Egy C program a következő részekből épülhet fel:

- fejlécfájlok beszerkesztése
- preproceszor konstansok, makrók definíciója
- globális változók, konstansok
- globális típusdefiníciók
- függvénydeklarációk
- függvénydefiníciók

Ezek közül bármelyik elhagyható, ha nincs rá szükség. Sorrendjük nem kötött, akár vegyesen is alkalmazhatók.

Fejlécfájlok beszerkesztése ► A *#include* kezdetű sorok, lásd az 5.2 fejezetben.

Preproceszor konstansok, makrók definíciója ► A *#define* kezdetű sorok, lásd a 31. fejezetben.

Globális változók, konstansok, típusdefiníciók ► Változókat, konstansokat, típusokat lehet definiálni függvényen kívül is. Ebben az esetben ezek globálisak lesznek, azaz bármely ezt követő függvényben használhatók. Használhatók továbbá a definíciót megelőzően vagy másik forrásfájlban is, ha előtte szerepel a deklarációjuk.

A globális változók kezdőértéke nulla, ha nem adunk kezdőértéket. (Emlékezzünk, hogy a függvényben definiált, azaz lokális **változó** esetén memóriaszemét volt!)

Korábbi programjainkban kerültük a globális változók használatát, és a jövőben is kerülni fogjuk, mert **globális változót**

használni veszélyes. Bár globális változót használni kényelmesnek tűnhet (nem kell paramétert átadni a függvénynek), a globális változók áttekinthetlenné és rugalmatlanná teszik a program felépítését. Ha egy változót bármely függvény megváltoztathat, akkor könnyen elveszíthetjük a fonalat, és csak hosszas keresgélés után deríthetjük fel, melyik függvény is változtatta meg számunkra kellemetlen módon. A globális változókkal operáló program később csak nehézkesen bővíthető.

Függvénydeklaráció, függvénydefiníció ► Lásd az 5.2 fejezetben.

Példa ►

```
#include <stdio.h>

#define INT_KIIR(a) printf("%d\n",a)

#include <stdlib.h>

void valtoztat(int);

int x;

extern int y;

int main(void) {
    int a=0,b;
    x=y;
    INT_KIIR(x);
    valtoztat(a);
    INT_KIIR(x);
    return 0;
}

int y=21;

void valtoztat(int y) {
    x=y;
}
```

Az *INT_KIIR* egy függvényszerű makró, mivel a preproceszor dolgozza fel, nincs pontosvessző a végén.

Az *x* változó kezdőértéke 0, mert globális, a *b* változó kezdőértéke memóriaszemét, mert lokális.

A *#include* sorok nem feltétlenül a fájl elején vannak. Oda tesszük, ahová be akarjuk szűrni a megadott fájl tartalmát (ad absurdum még függvényen belül is lehet speciális esetben).

A *main* függvény előtt szerepel a *valtoztat* függvény és az *y* változó deklarációja. Ezeket a *main* után definiáljuk. A deklarációban ha akarjuk, elhagyhatjuk a paraméterváltozó nevét, csak a típusát adjuk meg. Az *extern* kulcsszó globális változó deklarációját teszi lehetővé, azaz egy máshol definiált globális változó létezését jelenti be. Általában a máshol egy másik forrásfájlban van, erre utal az *external*=külső szóból eredő tárolási osztály. Lokális változó esetén nem használhatjuk az *extern* tárolási osztályt.

A *valtoztat* függvény definíciójában szerepel egy lokális *y* nevű változó definíciója. A lokális változó eltakarja a globálisat, azaz ebben a függvényben nem érhető el a globális változó, de más problémát a dolog nem okoz.

21.3 Láthatóság, tárolási osztályok

Az alábbiakban, ha fájlról vagy forrásfájlról beszélünk, azt a fájlértjük alatta, ami kijön a preproceszorból, és a fordítóhoz kerül. Azaz a *#include*-dal beszerkesztett fájlok vége nem jelenti a fájl végét.



Lokalitás ► A függvényen kívül definiált változók a globális változók, a függvényen belül definiált változók a lokális változók.

Láthatóság ► A globális változók a definíciótól a fájl végéig láthatók, a lokális változók az azokat definiáló blokk végéig, azaz a blokkot lezáró }-ig.

Lokális változó nem csak a függvény elején, hanem bármely {} blokk elején (a C99 szabványtól már nem csak az elején) definiálhatók. Például `if(x>y){int a=x; x=y; y=a; }`. Ebben az esetben az `a` változó csak az `if`-et lezáró }-ig él, utána megsemmisül, és persze nem is látható.

Élettartam ► Mikor jön létre a változó és meddig létezik. Az `auto` és `register` változók definíálásuktól a blokk végéig élnek; a globális, statikus és `extern` változók a program működése alatt mindvégig élnek; a dinamikus változók a lefoglalásuktól a felszabadításukig, de legkésőbb a program végéig élnek.

A tárolási osztály ► a változók ill. függvények tárolási módját határozza meg. Négy nevesített tárolási osztály van a C-ben: `auto`, `register`, `static`, `extern`.

- **auto, register:** ezek csak lokális változók lehetnek. Ha egy lokális változónál nincs megadva, a fordító dönti el, hogy a kettő közül melyiket választja. (Egyébként ha `register`-t adunk meg, a fordító akkor is automatikussá definiálhatja a változót.) Az `auto` és `register` változók a blokk/függvény végéig élnek, ott megsemmisülnek, kezdőértékük bármi lehet (memóriaszemét). A `register` változót a fordító a processzor valamelyik regiszterében próbálja elhelyezni, ezáltal gyors hozzáférést biztosítva.
- **static:** függvény és változó is lehet. Változó esetében mást jelent, ha lokális változóra adjuk meg, és mást, ha globálisra. Ha statikus változónak nem adunk kezdőértéket, akkor 0 lesz.
 - Ha egy lokális változó statikus, akkor a változó megőrzi az értékét a függvény két meghívása között. Olyan, mintha egy globális változót használnánk, ami más függvényből nem látható. A megadott kezdőértéket csak a függvény első meghívásakor kapja meg, később ezzel nem íródik felül (a lenti példában a `szamol` függvény által visszaadott érték minden olyan hívásnál eggyel nagyobb, amikor `x>0`).
 - Ha egy globális változó statikus, az nem érhető el másik forrásfájlból `extern` segítségével.
 - A függvények mindig globálisak. Ha egy függvény statikus, akkor nem érhető el másik forrásfájlból a deklaráció megadásával.
- **extern:** ugyanazt a célt szolgálja, mint a függvénynél a deklaráció. Tehát így lehet változót deklarálni úgy, hogy az adott helyen nem definiáljuk. Lokális változóra nem hivatkozhatunk `extern`-nel (a lenti programban nem írhatuk volna `extern int glob;` helyett, hogy `extern int a;`).

```
#include <stdio.h>

// csak ebben a forrásfájlban elérhető, 0 kezdőérték
static int sta_glo;

// ez csak deklaráció, kell definíció, ami
// másik fájlban is lehet
extern int kulso;

static void sta_fv(){
    printf("Csak itt elérhető.\n");
}

extern void ext_fv(); // = ext_fv();

int szamol(register int x){
    static int i=0; // Ha nem adunk kezdőértéket,
                  // akkor is biztosan 0.
                  // Csak az első hívásnál
```

```
        // kap értéket!
        if(x>0) i++;
        return i;
    }

int main(){
    auto int a=3; // = int a=3;
    register int b=5;
    auto int c; // memóriaszemét kezdőérték
    register int d; // memóriaszemét kezdőérték
    extern int glob; // csak deklaráció
    static int st; // 0 kezdőérték
    {
        int x; // auto változó, a blokk végéig él
    }
    // itt már nem írhatjuk, hogy x=3;,
    // mert a blokk után vagyunk
}

int glob; // másik forrásfájlból is elérhető
        // extern-nel, 0 kezdőérték
```

Gondolkozzunk együtt!

Oldja meg egyedül!

lib készítése is

22. A parancssor

Amikor a C nyelv megszületett, még nem léteztek grafikus felületű operációs rendszerek, csak szövegesek. Ezekben teljesen természetes volt a programok parancssori paraméterrel való indítása. Ez a lehetőség ma is megvan, bár ritkábban használjuk. A C nyelv a *main* függvény paramétereiként tudja átvenni a parancssori paramétereket.

Pl. `mkdir -p /home/laci/sajtok/gouda`

A fenti unix program létrehozza a megadott mappát, a `-p`-vel jelezzük, hogy ha a teljes útvonal nem létezik, akkor hozza létre a hiányzó mappákat is.

Próbáljuk meg elkészíteni a program C változatát!

mkdir.c első változat:

```
int main(void) {
    ...
    return 0;
}
```

Ez nem látja a parancssori paramétereket.

mkdir.c második változat:

```
int main(int db) {
    printf("A paraméterek száma: %d\n", db);
    ...
    return 0;
}
```

Ez már tudja, hogy hány paramétert adtunk meg, de azt még nem, hogy mik a paraméterek.

mkdir.c harmadik változat:

```
int main(int db, char * parameterek[]) {
    int i;
    printf("A program neve: %s\n", parameterek[0]);
    for (i=1; i<db; i++)
        printf("A %d. paraméter: %s\n", i, parameterek[i]);
    ...
    return 0;
}
```

Ez a változat kezeli a parancssori paramétereket. A példában szereplő paraméterekkel meghívva *db* értéke 3 lesz, a *parameterek* tömb pedig:

[0]	→	"mkdir"
[1]	→	"-p"
[2]	→	"/home/laci/sajtok/gouda"

Mivel a 0 indexű paraméter a program neve, *db* mindig legalább 1. A *main* változóinak neve bármi lehet, gyakran *argc*-nek hívják a darabszámot és *argv*-nek a pointertömböt. Ha számot kérünk parancssori paraméterként, a program azt is sztring formában kapja. A sztringből pl. a *scanf* függvény segítségével tudunk számot nyerni (az *atoi*, *atol* függvények is használhatók).

Néhány C fordító további *main* formátumot is megenged. a Visual C az alábbi:

```
int main(int argc, char *argv[], char *envp[]);
```

Az *envp* az operációs rendszer ún. környezeti változóit tartalmazza, a tömb végét NULL pointer jelzi.

Gondolkozzunk együtt!

Oldja meg egyedül!

23. Fájlkezelés

A fájlok arra szolgálnak, hogy változóink tartalmát a számítógép kikapcsolása után is megőrizhessük, illetve hogy mások által elmentett adatokat feldolgozhassunk, mások által készített programokat futtathassunk. Ezekben ugyanolyan bajtokat tárolunk, mint a memória tartalma. Nem kell tehát mást tenni, mint a memória tartalmát változtatás nélkül a háttértárra másolni, illetve onnan beolvasni. A C szabványos függvényeivel ez nagyon egyszerű.

A C nyelv **kétféle** magas szintű fájlkezelést támogat: **egy általános fájl és egy speciális fájlkezelést**.

Az általános fájlkezelést **bináris fájlkezelésnek** nevezzük. Ha egy fájl binárisan kezelünk, akkor a fájlkezelő függvénynek egy memóriacímet adunk, és egy egész számot, mely megmondja, hogy hány bajtot szeretnénk beolvasni vagy kiírni.

A speciális fájlkezelés alatt azt értjük, hogy a nyelv ismeri az adott fájl típus belső felépítését, azt tehát a programozónak nem kell. A C nyelvben támogatott speciális fájl a szöveges fájl. Más programnyelvek (pl. php) támogatja például a jpg, png, stb grafikus állományokat, a zip archívumokat, stb. A C sajnos nem ilyen bőkezű, bár léteznek külső fejlesztésű könyvtárak ezen fájl típusok támogatására.

Bárhogyan is kezeljük a fájl, azt meg kell nyitni használat előtt, amit az **fopen** függvénnyel tehetünk, és be kell zárni, ami az **fclose** függvény dolga.

Az **fopen** prototípusa:

```
FILE *fopen(const char *filename, const char *mode);
```

A függvény tehát egy pointert ad vissza, melynek típusa **FILE ***. A **FILE** egy struktúratisz, amely az **stdio.h**-ban van definiálva. A struktúrában található az az adatok, melyek segítségével a szabványos függvények a fájlal dolgozni tudnak. Számunkra nincs jelentősége a struktúra tartalmának. Annyi a dolgunk, hogy az **fopen** által visszaadott pointert odaadjuk a fájlból olvasó vagy oda író függvényeknek, melyek a fájlpointer alapján tudják, mely fájlal kell dolgozniuk.

Az **fopen** első paramétere a fájl nevét tartalmazó sztring, melyet konstansként vagy karaktertömbben egyaránt megadhatunk. A fájl neve elérési útvonalat is tartalmazhat. Ha nem adjuk meg, az aktuális mappában keresi a fájl.

Az **fopen** második paramétere szintén sztring, a fájl megnyitásának módja. A mód első karaktere 'w', 'r' vagy 'a', jelentése „write” „read” ill. „append” azaz írni, olvasni vagy hozzáírni akarunk-e a fájlhoz. Második karaktere 'b' vagy 't': „binary” vagy „text”. A 't' elhagyható, tehát pl. a "wt" és a "w" ugyanazt jelenti. Ha írásra nyitjuk meg a fájl, akkor a fájl korábbi tartalma elvész, ha hozzáírásra, akkor a korábbi tartalom megmarad, és íráskor az eddigi rész után kerül az új tartalom!



A második paraméter esetén további lehetőségek is vannak, ezek ismerete nem kötelező:

- "w+", "r+", "a+" is használható (azaz "w+t", "w+b" stb.). Itt a „+” azt jelenti, hogy a fájl írható és olvasható is egyszerre. A három változat között az a különbség, hogy a „w+” letörli a fájl, ha létezett korábban, és a megnyitott üres fájl elején állunk; „r+” esetén szintén a fájl elején állunk, viszont nem törlődik a korábbi tartalom; „a+” esetén nem törlődik a tartalom, és a fájl végén állunk.
- A "w+", "r+", "a+" opciókat akkor tudjuk kihasználni, ha tudunk mozogni a fájlban. A **rewind** függvénnyel a fájl elejére mozoghatunk. Pontosabb pozicionálást tesz lehetővé az **fseek** függvény. Pl.: **fseek(fp, n, SEEK_SET)**; Első paramétere a fájlpointer, második, hogy hány bajtnyt akarunk mozogni, harmadik paramétere három konstans lehet: **SEEK_SET**, **SEEK_CUR** és **SEEK_END**. **SEEK_SET** esetén a fájl elejétől mozogunk, **SEEK_CUR** esetén a jelenlegi helytől, **SEEK_END** esetén a fájl végétől. Az **n** értéke negatív is lehet, ekkor a megadott helytől ennyi bajtot mozogunk visszafelé. Az **n long int** típusú érték, ami 32 bites rendszerben általában 32 bites, ami maximum 2 GB-os tartomány elérését teszi lehetővé, nagy fájlknál erre figyeljünk!
- Az **ftell** függvény megmondja, hogy a fájl elejétől hány bajtra vagyunk jelenleg. Ez is **long** típusú értéket ad vissza, tehát 2 GB fölött nem működik 32 bites **long** típus esetén.

Ha nem sikerült megnyitni egy fájl valamiért (pl. nem létező fájl akarunk olvasni, vagy nem írható meghajtóra akarunk fájl írásra nyitni), akkor **fp** értéke **NULL** lesz.

Miután végeztünk a fájl írásával, be kell zárni azt az **fclose** függvény segítségével. Az **fclose** egy paramétert kap: a megnyitott fájl fájlpointerét. Tilos egy már bezárt fájlra ismét meghívni az **fclose** függvényt!

23.1 Bináris fájlok

Fájl bináris olvasása az **fread** függvénnyel, írása az **fwrite** függvénnyel történik. A két függvény paraméterezése és visszatérési értéke ugyanaz.

- Első paraméter: a kiírandó/olvasandó adat memóriacíme.
- Második paraméter: egy adat (egy tömbelem) mérete.
- Harmadik paraméter: a tömb elemszáma (nem tömbnél 1).
- Negyedik paraméter: a fájlpointer.
- Visszatérési érték: hány darab tömbelemet sikerült kiírni/beolvasni. Ha ismeretlen méretű fájl olvasunk be, akkor ezt használhatjuk annak vizsgálatára, hogy elértük-e a fájl végét.

A második és harmadik paraméter szorzata a tömb méretét adja bajtokba, hasonlót láttunk a **calloc** függvényénél, ott azonban a két paraméter fordított sorrendben szerepelt: előbb az elemszám, aztán egy elem mérete. Ilyen az, amikor két ember készít el egy programozási nyelvet ;-).

Példaprogram:

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

void hiba(char * s){
    fprintf(stderr, "%s\n", s);
    exit(1);
}

typedef struct{
    char nev[40], cim[100];
    double egyenleg;
}elofizeto;

int main(void){
    char szoveg[100]="Nyaktekerészetimellfekvenc.";
    double t[20];
    int x=3;
    elofizeto jozsi={"Józsi", "Híd alatt", 2300};
    elofizeto et[30];

    //fájlnyitás

    FILE * fp=fopen("progalapjai.xyz", "wb");
    if(fp==NULL)
        hiba("Fajlnyitas.");

    //fájlírás

    if(fwrite(szoveg, 1, 100, fp)!=100)
        hiba("Nem írta ki jól a char tombot.");
    if(fwrite(t, sizeof(double), 20, fp)!=20)
        hiba("Nem írta ki jól a double tombot.");
    if(fwrite(&x, sizeof(int), 1, fp)!=1)
        hiba("Nem írta ki jól az int-et.");
    if(fwrite(&jozsi, sizeof(elofizeto), 1, fp)!=1)
        hiba("Nem írta ki jól jozsit.");
    if(fwrite(et, sizeof(elofizeto), 30, fp)!=30)
        hiba("Nem írta ki jól az elofizeto tombot.");

    //fájlzárás, fájlnyitás

    fclose(fp);
    fp=fopen("progalapjai.xyz", "rb");
    if(fp==NULL)
        hiba("Fajlnyitas 2.");

    //fájlolvasás

    if(fread(szoveg, 1, 100, fp)!=100)
```

```

        hiba("Nem olvasta be jól a char tombot.");
    if (fread(t, sizeof(double), 20, fp) != 20)
        hiba("Nem olvasta be jól a double tombot.");
    if (fread(&x, sizeof(int), 1, fp) != 1)
        hiba("Nem olvasta be jól az int-et.");
    if (fread(&jozsi, sizeof(elfozeto), 1, fp) != 1)
        hiba("Nem olvasta be jól jozsit.");
    if (fread(et, sizeof(elfozeto), 30, fp) != 30)
        hiba("Nem olvasta be jól az elfozetombot.");

//fájlzárás

fclose(fp);
return 0;
}

```

A példaprogram hibakezeléstől megcsupaszított változata (ez áttekinthetőbb, de a gyakorlatban fontos a hibakezelés, ne hagyjuk el!):

```

#include <stdio.h>

typedef struct {
    char nev[40], cim[100];
    double egyenleg;
} elfozeto;

int main(void) {
    char szoveg[100] = "Nyaktekereszetimellfekvenc.";
    double t[20];
    int x=3;
    elfozeto jozsi = {"Józsi", "Hid alatt", 2300};
    elfozeto et[30];

//fájlnyitás

FILE * fp = fopen("progalapjai.xyz", "wb");

//fájlírás

fwrite(szoveg, 1, 100, fp);
fwrite(t, sizeof(double), 20, fp);
fwrite(&x, sizeof(int), 1, fp);
fwrite(&jozsi, sizeof(elfozeto), 1, fp);
fwrite(et, sizeof(elfozeto), 30, fp);

//fájlzárás, fájlnyitás

fclose(fp);
fp = fopen("progalapjai.xyz", "rb");

//fájlolvasás

fread(szoveg, 1, 100, fp);
fread(t, sizeof(double), 20, fp);
fread(&x, sizeof(int), 1, fp);
fread(&jozsi, sizeof(elfozeto), 1, fp);
fread(et, sizeof(elfozeto), 30, fp);

//fájlzárás

fclose(fp);
return 0;
}

```

Látható, hogy egy bonyolult struktúrát is egyszerűen kiírt és beolvasott. Szöveges fájl esetén az adatokat külön-külön ki kellett volna írni/be kellett volna olvasni. A bináris fájlban az adatok abban a bináris formában kerülnek kiírásra, ahogy a számítógép memóriájában megtalálhatók, tehát az így létrejött fájl általában nem olvasható szövegszerkesztő programmal.

Ezzel a módszerrel a szöveges fájlok is olvashatók, például karakterenként. Ebben az esetben persze nekünk kell figyelniük például az újsor karakterekre, és ha a szöveges fájlban számok is vannak, akkor nekünk kell a külső formátumú (karakterekből álló) számot belső formátumúra (binárisra) alakítani.

23.2 Szöveges fájlok

A szöveges fájlok, ahogy nevünk is mondja, szövegeket tartalmaznak. Bármely szövegszerkesztővel megnyithatók, és különböző számítógép-architektúrák között hordozhatóak.

(Bináris fájlok esetében tisztában kell lenni az adatok tárolási módjával, ha különböző rendszerek között akarjuk átvinni az adatokat. Például az Intel/AMD processzorok az egy bájtnál nagyobb adattípusok bájtaikat fordított sorrendben tárolják. Erről meggyőződhetünk, ha lefuttatjuk az alábbi kódot:

```

#include <stdio.h>

int main(void) {
    int a = 0x04030201;
    char * p = (char*) &a;
    printf("%x, %x, %x, %x\n", p[0], p[1], p[2], p[3]);
    return 0;
}

```

Intel architektúrán az eredmény: 1,2,3,4. Pl. Motorola architektúrán viszont 4,3,2,1.)

A szöveges fájlok hátránya, hogy elsősorban a beolvasásuk sokkal nehezekebb, mint a bináris fájloké. Ha bináris fájlba mentünk egy egész számot, az mindig ugyanannyi bájtot foglal, míg szöveges formában nem ez a helyzet: a 3 és a -21342131 egyaránt befér egy 32 vagy több bites int-be, viszont láthatóan az előbbi egy bájtot (karaktert), utóbbi viszont 9 bájtot foglal szöveges formában. Ha egy karaktertömböt írunk ki bináris fájlba, az mindig ugyanolyan hosszú, ha egy sztringet írunk ki szöveges fájlba, az nem egyforma hosszú.

További gond, hogy mivel a szöveges fájl bárki bármilyen szövegszerkesztővel szerkeszthető, fel kell készülni arra, hogy a beolvasott fájl nem pont olyan, mint a kiírt: a felhasználó esetleg két szóközt írt az adatok közé egy helyett, stb.

A szöveges fájlok kezelése mindenkinek ismerős lesz, mert a képernyőre író és a billentyűzetről olvasó függvényekhez nagyon hasonló függvényeket használhatunk.

A .c ill. .h kiterjesztésű forrásállományok éppúgy szöveges fájlok, mint a .html vagy a .php fájlok.

A függvények ►

fprintf: a *printf* és az *sprintf* rokona, szöveget ír szöveges fájlba. Első paramétere a megnyitott fájl fájlpointerre, ezt követi a formátumsztring, majd a formátumsztringben jelzett változók illetve konstansok.

fputs: a *puts* fájlba író változata. Két paramétere van: a kiírandó sztring és a fájlpointer. A *puts*-sel ellentétben az *fputs* nem ír ki újsort a sztring végén.

fputc: a *putchar* fájlba író változata. Két paramétere van: a kiírandó karakter és a fájlpointer.

Például: Írjunk programot, amely kiírja a „hellovilag.c” nevű fájlba a „Hello világ!” szöveget kiíró C programot!

```

#include <stdio.h>

int main(void) {
    FILE * fp = fopen("hellovilag.c", "wt");
    if (fp != NULL) {
        fprintf(fp, "#include <stdio.h>\n\n");
        fprintf(fp, "int main(void) {\n");
        fputs("    printf(\"Hello világ!\");\n", fp);
        fputc('}', fp);
        fputc('\n', fp);
        fclose(fp);
    }
    else {
        fprintf(stderr, "Sikertelen fajlnyitsa.\n");
    }
    return 0;
}

```

Fordítsa és futtassa le a programot, majd keresse meg a létrejött „hellovilag.c”-t!

fscanf: a *scanf* és az *sscanf* rokona, szöveget olvas szöveges fájlból. Első paramétere a megnyitott fájl fájlpointerre, ezt követi a formátumsztring, majd a formátumsztringben jelzett változók címei.

fgets: a *gets* fájlból olvasó változata. Három paramétere van a 12.2 fejezetben írottaknak megfelelően: a cél karaktertömb címe, a karaktertömb mérete és a fájlpointer.

fgetc: a *getchar* fájlból olvasó változata. Egy paramétere van: a fájlpinter, visszatérési értéke a beolvasott karakter, vagy *EOF*, ha nem sikerült az olvasás.

Például írjunk programot, amely beolvassa a „hellovilag.c” fájl tartalmát, és kiírja a képernyőre!

```
#include <stdlib.h>

void hiba(const char * s){
    fprintf(stderr,"%s",s);
    exit(1);
}

int main(void){
    FILE * fp=fopen("hellovilag.c","rt");
    char s[100];
    int ch;

    if(fp!=NULL){
        if(fgets(s,100,fp)==NULL)
            hiba("1. sor hiányzik");
        printf("%s",s);
        if(fgets(s,100,fp)==NULL)
            hiba("2. sor hiányzik");
        printf("%s",s);
        if(fscanf(fp,"%s",s)!=1)
            hiba("int hiányzik");
        printf("%s ",s);
        if(fscanf(fp,"%s",s)!=1)
            hiba("main(void){ hiányzik");
        printf("%s",s);
        if(fgets(s,100,fp)==NULL)
            hiba("ujrsor jel hiányzik (1)");
        printf("%s",s); // az ENTER-t olvassa
        if(fgets(s,100,fp)==NULL)
            hiba("4. sor hiányzik");
        printf("%s",s);
        if((ch=fgetc(fp))==EOF)
            hiba("} hiányzik");
        putchar(ch);
        if((ch=fgetc(fp))==EOF)
            hiba("ujrsor jel hiányzik (2)");
        putchar(ch); // az ENTER-t olvassa
    }
    else{
        fprintf(stderr,"Sikertelen fajlnyitsa.\n");
    }
    return 0;
}
```

Mindhárom függvény használatát be szeretnénk volna mutatni, ahogy az előző példában is, de egyszerűbben is megoldhattuk volna a feladatot, pl. így:

```
int main(void){
    FILE * fp=fopen("hellovilag.c","rt");
    char s[100];

    if(fp!=NULL){
        while(fgets(s,100,fp)!=NULL)printf("%s",s);
    }
    else{
        fprintf(stderr,"Sikertelen fajlnyitsa.\n");
    }
    return 0;
}
```

Excel fájl feldolgozása ► Magyar nyelvű Excellel készítettünk egy táblázatot, melyben a következő adatszlopok szerepelnek:

- Név
- Anyja neve
- Születési év.hónap.nap, pl.: 1848.03.15
- Telefonszám

A táblázatot elmentjük az „emberek.csv” fájlba. (A magyar Excel pontosvesszővel tagolt fájl hoz létre, míg az angol vesszővel tagoltat, ahogy a csv nevéből is következik: comma separated values – vesszővel elválasztott értékek.)

	A1		Gipsz Jakab		
	A	B	C	D	E
1	Gipsz Jakab	Trutty Vilma	1848.03.15	+36-30-112345678	
2	Gipsz Jakabné	Mocsári Vipera	1988.01.03	+36-30-112345678	
3					

Csv-be mentve a következő szöveges fájlt kapjuk:

```
Gipsz Jakab;Trutty Vilma;1848.03.15;+36-30-112345678
Gipsz Jakabné;Mocsári Vipera;1988.01.03;+36-30-112345678
```

Feladat: írjuk ki azon a személyek nevét, akik márciusban születtek, anyjuk második neve (ami valószínűleg a keresztnéve) pedig V-vel kezdődik!

Ha a szöveges fájlunk ilyen összetett sorokat tartalmaz, akkor nem érdemes az *fscanf*-fel bajlódni, jobban járunk, ha soronként olvassuk, és a sort mint sztringet dolgozzuk fel.

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

void hiba(const char * s){
    fprintf(stderr,"%s",s);
    exit(1);
}

// A forrás sztringet szeparátor karakterek osztják
// oszlopokra.
// Az oszlopadik oszlopot átmásolja a cél tömbbe
int get_oszlop(const char * forras
               ,char szeparator,int oszlop,char * cel){
    int n,i,j;

    // az n. oszlop elejének megkeresése
    for(i=n=0; forras[i]!=0 && n<oszlop; i++)
        if(forras[i]==szeparator)n++;
    if(n!=oszlop)return 0; // nincs n. oszlop

    // átmásolás a cél tömbbe
    for(j=0; forras[i]!=0 && forras[i]!=szeparator;
        i++,j++)cel[j]=forras[i];
    cel[j]=0;
    return 1;
}

int main(void){
    FILE * fp=fopen("emberek.csv","rt");
    char s[400],t1[100],t2[100];

    if(fp==NULL)hiba("Sikertelen fajlnyitsa.\n");

    // soronként olvasunk
    while(fgets(s,400,fp)!=NULL){

        // t1-be az anyja neve, t2-be a keresztnév
        if( get_oszlop(s,';',1,t1)
           && get_oszlop(t1,' ',1,t2) ){

            // Ha a keresztnév V-vel kezdődik,
            // t1-be teszi a születési dátumot
            if(t2[0]=='V' && get_oszlop(s,';',2,t1)){
                int ev,honap,nap;

                // Ha márciusi, kiírja
                if(sscanf(t1,"%d.%d.%d"
                        ,&ev,&honap,&nap)==3 && honap==3){
                    get_oszlop(s,';',0,t2);
                    printf("%s\n",t2);
                }
            }
        }
    }
    return 0;
}
```


A `get_oszlop` függvény a forrás sztringből a cél sztringbe másolja azt a részt, amely a szeparátor karakter `oszlop`-adik és `oszlop+1`-edik előfordulása között található. Ha nincs benne `oszlop`-szor a szeparátor, HAMIS visszatérési értékkel jelzi ezt. Helyesen kezeli azt az esetet is, amikor van `oszlop` darab `oszlop`, de ez az utolsó, utána nincs szeparátor.

A `main` függvényben egy `while` ciklus olvassa az „emбекek.csv” sorait a fájl végéig. A feldolgozás során végig figyelünk a hibákra. Ha egy sor hibás, azaz nincs benne annyi `oszlop`, amennyi kell, a következő sorra lépünk.

A `get_oszlop` függvénnyel kiolvassuk `t1`-be az 1-es oszlopot, azaz az anyja nevét, és abból kiolvassuk a második nevet a `t2`-be. Ez azért lehetséges, mert az oszlopokat pontosvessző, a neveket pedig szóköz választja el egymástól.

Az oszlopokat 0-tól számozzuk, ha gondolja, átírhatja a programot úgy, hogy 1-től számoljon. Mit kell ehhez változtatni?

Figyeljük meg, hogy a `get_oszlop(s,',' ,1,t1) && get_oszlop(t1,' ,1,t2)` kifejezésben kihasználjuk az `&&` operátor mindkét speciális tulajdonságát: a rövidzár és azt, hogy kiértékelési pont. A rövidzár azt jelenti, hogy ha az első `get_oszlop` hívás HAMIS-sal tér vissza, a második hívás nem történik meg, mert HAMIS `&&` BÁRMI eredménye biztosan HAMIS. Kiértékelési pont azt jelenti, hogy az `&&` előtti kifejezés, azaz a függvényhívás biztosan megtörténik, mielőtt az `&&` utáni kifejezés, azaz a második függvényhívás megtörténne. Ez azért fontos, mert a második függvényhívás azt a tömböt használja, amit az első előállított. Kiértékelési pontként nem működő operátorok esetében ez nem garantált, azaz ha pl. `int x=get_oszlop(s,',' ,1,t1) + get_oszlop(t1,' ,1,t2);` utasítást írunk, akkor lehetséges, hogy a második függvényhívás előbb történne, mint az első, ami így hibás bemeneti tömböt kapna.

Az `scanf` visszatérési értéke a sikeresen beolvasott változók száma, ami itt három kell legyen. Ebben az `if`-ben is kihasználjuk az `&&` speciális tulajdonságait.

23.3 Szöveges fájlok és bináris fájlok összevetése

A szöveges fájl előnye, hogy platformfüggetlen: teljesen más számítógép architektúrán is ugyanúgy olvasható. Ha a fenti bináris fájl példát 32 bites és 16 bites operációs rendszer alatt is lefordítjuk, akkor egyik esetben 32 bites az egész szám, a másik esetben 16 bites. Ha az egyik rendszerben elmentett fájl a másikban akarjuk olvasni, gondjaink lesznek emiatt. A szöveges fájlal nincs ilyen probléma.

A szöveges fájl előnye, hogy minden gépen van egyszerű szövegszerkesztő program, mellyel a felhasználó módosítani tudja a fájl tartalmát.

A szöveges fájl hátránya, hogy minden gépen van egyszerű szövegszerkesztő program, mellyel a felhasználó módosítani tudja a fájl tartalmát. Emiatt beolvasáskor mindig figyelni kell arra, hogy helyes-e a fájl tartalma. Bináris fájlnál nyugodtan lehet feltételezni, hogy minden adat a helyén van.

A bináris fájl előnye, hogy mivel nem kell konvertálni külső és belső formátum között, jóval gyorsabban írható/olvasható, mint a szöveges.

A bináris fájl előnye, hogy tömbök, struktúrák nagyon egyszerűen írhatók és olvashatók. Szöveges fájlba közvetlenül csak alaptípusokat tudunk írni, tehát a tömbök elemeit egyenként (pl. ciklussal), a struktúrákat tagonként.

A szöveges fájl hátránya, hogy beolvasása nagyon macerás, bináris fájlal egy adattípus mindig ugyanakkora helyet foglal. Ha egy egész számot szöveges formátumban írunk ki, akkor az lehet pl. 2, vagy lehet pl. -876498290 is. Előbbi 1 karakternyi helyet foglal, utóbbi 10-et. A sztringek helyzete még bonyolult-

tabb: elméletileg bármilyen karakter előfordulhat benne (akár soremelés is), mi legyen az elválasztó karakter? Bináris fájl esetén a sztringekkel sincs semmi gond, ahogy a fenti példában szerepel: elmentjük az egész tömböt, azt a részt is, ahol már véget ért a sztring. Ezzel ugyan több helyet foglalunk, de cserébe nagyon egyszerű a kezelés.

Szöveges fájl és bináris fájl közötti döntésnél mindig az adott feladatnak legjobban megfelelő formátumot válasszuk!

23.4 Szabványos bemenet és kimenet

A program szabványos bemenete alapértelmezés szerint a billentyűzet, kimenete a képernyő, de ez átírható.

Pl.: `programom.exe > kimenet.txt`

Létezik három olyan szabványos globális változó, amely szöveges fájlként viselkedik:

```
stdin // standard input
stdout // standard output
stderr // standard error
```

Ezek közül az `stdin` olvasható, az `stdout` és `stderr` írható a szövegfájl-kezelő függvényekkel.

Megnyitni, bezárni ezeket nem kell és nem is szabad.

Használat pl.:

```
fprintf(stdout, "Hogy hívnak? ");
if (fgetc(s, 100, stdin) == NULL) {
    fprintf(stderr, "Sikertelen beolvasás.");
    exit(1);
}
fprintf(stdout, "%s\n", s);
```

A szabványos kimenet és a szabványos hibakimenet külön-külön átírható.

Gondolkozzunk együtt!

Oldja meg egyedül!

III. RÉSZ: HALADÓ PROGRAMOZÁS

24. Dinamikus önhivatkozó adatszerkezetek 1. – láncolt listák

Sok adatot feldolgozó programok esetében a feladat jellegétől függően a következő esetek fordulhatnak elő:

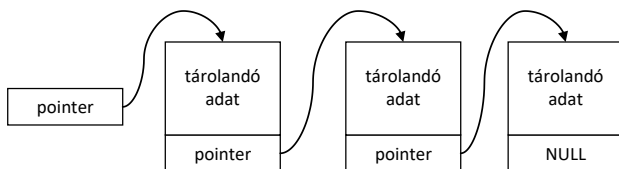
- Az adatok eltárolására nincs szükség, mert egyszer végigolvasva az adatokat a kívánt eredményre jutunk. Például ha meg kell számolni a bemenetről érkező szövegben a kisbetűket.
- Ha a program lefordításának időpontjában tudjuk, hogy pontosan vagy maximum mennyi adatra számíthatunk, amit el is kell tárolni, akkor tömböt alkalmazhatunk. Például konstans sztring; az ötös lottó nyerőszámai, háromdimenziós vektorok és műveleteik, stb.
- Ha egy program futása során az adatok beolvasása előtt tudjuk meg, hogy pontosan vagy maximum mennyi adatra számíthatunk, amit el is kell tárolni, akkor dinamikus tömböt használunk. Például két tetszőleges mátrix szorzását elvégző függvény az eredményt egy dinamikus tömbben adja vissza.
- Ha nem tudjuk előre, mennyi adattal kell majd dolgozunk, vagy más okból egyszerűbb, ha mindig az igényeinknek megfelelő mennyiségű memóriát tartunk lefoglalva. Például ismeretlen mennyiségű mérési adat feldolgozása.

A memóriakezelés összes C eszközét megismertük már korábban, tehát az ismeretlen mennyiségű adat eltárolását ezekkel az eszközökkel kell megoldanunk.

Az egyik lehetséges megoldás, hogy lefoglalunk egy dinamikus tömböt, majd ha betelik, foglalunk egy nagyobb, ahová átmásoljuk a kicsit, és aztán letöröljük az eredetit. Ezzel az a probléma, hogy sok felesleges adatmozgatással jár, és amíg mindkét tömb létezik, sok felesleges memóriát foglalunk. (A *realloc* függvény ugyanezt teszi.)

A másik megoldás, hogy egyenként foglalunk memóriát az érkező adatoknak. A memória lefoglalása a *malloc/calloc* függvények valamelyikével történik, melyek visszaadják a lefoglalt terület címét. Ezt a címet meg kell jegyeznünk, hogy megtaláljuk a lefoglalt memóriaterületet. Ez azt jelenti, hogy ha ezerszer foglalunk helyet, ezer címet kell tárolnunk, ha tízezerszer foglalunk, akkor tízezret... Vagyis ugyanúgy nem tudjuk, mennyi címet kell tárolnunk, mint ahogy azt sem, hogy mennyi adatot, viszont ez a két szám megegyezik. Megoldás: tároljuk struktúrában egy adatot és egy címet. A struktúrában tárolt pointer nem saját maga mutat, hanem a következő adatra.

24.1 Egyszerű láncolt lista



Kezdetben csak egy pointerünk van, aztán lefoglalunk egy struktúrát, és eltároljuk benne a beolvasott adatot. A pointert erre állítjuk. A struktúra pointerét NULL-ra állítjuk jelezve, hogy itt a lista vége. Amikor lefoglalunk egy újabb adatot, ezt a pointert az új adatra állítjuk, és az új adat pointerre lesz NULL.

Tehát mindig lesz szabad pointerünk, amivel mutathatunk a következő adatra.

A struktúrában lévő pointer ugyanolyan struktúrákra mutat, mint saját maga → önhivatkozó adatszerkezet.

Az ilyen felépítésnek az egyszerű bővíthetőségen túlmenően más előnyei is vannak:

- Ha a lista közepére akarunk új adatot betenni, akkor csak két pointert kell átállítani, míg ha pl. egy tömb közepére szeretnénk beszúrni, a beszúrás utáni összes adatot eggyel hátrébb kell másolni.
- Ha egynél több pointer van a struktúrában, nagyon változatos adatszerkezeteket hozhatunk létre.

A hátrány, hogy ha a lista valamelyik elemét szeretnénk megkapni, ahhoz végig kell járni a lista elejétől az összes elemet, nem lehet egyszerűen sorszám alapján kiolvasni, mint egy tömbnél.

Minta feladat ▶ Írjon programot szabványos C nyelven, amely beolvassa egy síkbeli poligon csúcsainak valós értékű koordinátáit a szabványos bemenetről, majd írja ki a súlyponttól jobbra található koordinátákat! A koordinátasorozat végét a 0.0 0.0 koordinátapár jelzi. A koordinátákat szóköz választja el egymástól.

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

typedef struct koordinata{
    double x,y;
    struct koordinata *next; // a köv. elemre mutat
} koord;

int main(){
    koord *start=NULL,*p; // starttól indul a lista
    double x,y;
    unsigned db=0;

    // A lista létrehozása

    scanf("%lg %lg",&x,&y);
    while(x!=0.0 || y!=0.0){
        koord *uj=(koord*)malloc(sizeof(koord));
        if(uj==NULL) return 1;
        uj->x=x;
        uj->y=y;
        uj->next=start; // a lista elejére fűzzük az
        // újat, az = akkor is jó, ha üres a lista
        start=uj;
        db++;
        scanf("%lg %lg",&x,&y);
    }

    // A koordináták összegzése

    for(x=y=0.0, p=start; p!=NULL; p=p->next){
        x+=p->x;
        y+=p->y;
    }
    x/=db; y/=db; // ez a súlypont, bár y-ra
    // igazából nincs is szükség

    // a súlyponttól jobbra esők kiírása

    for(p=start; p!=NULL; p=p->next){
        if(p->x>x) printf("(%g,%g)\n",p->x,p->y);
    }

    // a lista törlése

    if(start!=NULL){
        for(p=start, start=start->next;
            start!=NULL; p=start, start=start->next)
            free(p);
        free(p);
    }
    return 0;
}
```

A fenti kódban új nyelvi elem a **nyíl operátor** (->). Ez az operátor egy pointer által mutatott struktúra valamelyik adattagját adja. Például az `uj->x` a `(*uj).x` helyett került a kódba, azzal tökéletesen azonos jelentésű. A `*uj` azért zárójellezendő, mert a `.` precedenciája magasabb, mint az indirekció operátoré (`*`). Mivel ez egy gyakori művelet a C nyelvben, a nyelv készítői úgy látták jónak, hogy a négy karakter (két zárójel, `*` és `.`) helyett csak két karakter kelljen a művelethez (mínusz: `-` és nagyobb: `>`).

A struktúrátípus létrehozásakor a struktúrán belül nem használhatjuk a `koord` típust, hiszen a név később szerepel a kódban, ezért itt még `struct koordinata` típust adunk meg. Később azonban már használható a rövidebb változat (természetesen a hosszabb is).

Megjegyzés: a struktúrán belül létrehozhatunk önmagára mutató pointert, hiszen az csak egy pointer, de nem hozhatunk létre `struct koordinata` típusú változót, mert az azt jelentené, hogy azon belül is van egy ilyen struktúra, meg azon belül is, meg azon belül is... A rekurzív definíció nem megengedett.

A koordinátákat ideiglenes változóba tesszük el, és csak akkor foglalunk új memóriát, ha nem az adatsor végét jelző 0,0 párra bukkantunk. Az újonnan lefoglalt struktúrába bemásoljuk a koordinátákat, majd a `next` pointert a lista kezdőelemére állítjuk (üres listánál NULL pointer: a lista végét a NULL pointer jelzi, úgyhogy ez így teljesen jó), a kezdőelem pedig ettől kezdve az újonnan lefoglalt struktúra lesz, azaz a lista elejére fűztük be az új elemet, más szóval az elemek fordított sorrendben kerültek a listába a beolvasáshoz képest.

Ha ez gondot okoz, megoldható, hogy a lista a beolvasás sorrendjével azonos sorrendű legyen, ehhez azonban egy újabb segédváltozó kell, ami a lista végére mutat, és a kód is valamivel bonyolultabb lesz. Ennek megvalósítását az olvasóra bizzuk.

Ugyancsak lehetőség, hogy a listát valamilyen feltételnek megfelelően ne a beolvasás sorrendjében, hanem más sorrendben építsük fel, például a kulcsnak választott elem növekvő vagy csökkenő sorrendjében. Erre látunk példát a következő szakaszban.

A fenti példában a listát `for` ciklussal járjuk be. A bejárás hasonlít egy tömb elemeinek végigjárásához: a ciklusváltozó ezúttal pointer, amit a kezdőelemre állítunk. A ciklus vége, ha NULL pointerhez érünk, a léptetés a ciklusváltozó következő elemre mozgatását jelenti.

A lista törlése igényel némi átgondolást: a segédváltozó mutasson a lista kezdőelemére, a kezdőelem pedig lépjen a következőre. Ekkor törölhetjük a segédváltozó által mutatottat, így a lista valóban egy elemmel rövidebb lesz. Ezt addig folytatjuk, míg van elem a listában. Amire figyelni kell, hogy `NULL->next` a program azonnali elszállását okozza, tehát ezért van az `if (start!=NULL)`, és ezért van a ciklus után az utolsó `free`.

14.2 Láncolt lista strázásával, függvényekkel, beszúrással

Az előző példa a lehető legegyszerűbb esetet mutatta be. Ha bonyolultabb a listakezelés, célszerű minden műveletnek külön függvényt létrehozni.

Ha az egyszerű listaépítésnél bonyolultabb műveletek is vannak, például beszúrással vagy elemcsere, akkor másképp kell eljárni, ha a lista belsejében kell változtatni, a lista elején kell változtatni vagy a lista végén kell változtatni.

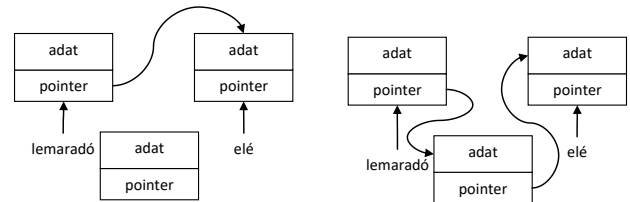
Ha a lista elejére szúrunk, a `start` pointert is meg kell változtatni, a meglévő listának viszont nem kell elemét megváltoztatni. Ha a lista végére szúrunk, az új elem pointere NULL lesz, a lista utolsó elemének pointere pedig az új elem címe. Ha közbe szúrunk, akkor a lista egy elemének pointere változik, az új elem

pointere pedig az adott elem utáni elem lesz. Ez megvalósítható, de a dolgunkat leegyszerűsítjük úgynevezett **strázsa** (dummy) elem(ek) alkalmazásával a lista egyik, vagy mindkét végén.

A strázsa olyan elem, amely nem tartalmaz hasznos adatot, így ha a lista elején áll, akkor elé biztos nem kell beszúrni semmit, ha pedig a végén áll, akkor utána nem kell beszúrni. Ha két strázsa van, akkor csak olyan beszúrással kell megírni a beszúró függvényt, ami két elem közé szúr be.

A következő példaprogramban a lista mindkét végén strázsa áll. A két strázst nem dinamikusán foglalt struktúra alkotja (lehetne dinamikus is).

Elem beszúrása a listába a következőképpen történik:



Megkeressük azt az elemet, amely elé szeretnénk beszúrni az új elemet. Ezért elé, mert ha rendezett listát akarunk építeni, akkor úgy találjuk meg a beszúrási helyét, hogy addig megyünk a listán, míg meg nem találjuk az első olyan elemet, amelynek kulcsa nagyobb (vagy kisebb, ha csökkenő a lista), mint a beszúrni kívánt elem kulcsa. **(Kulcs=az az adat, amely szerint rendezzük a listát.)** Egy irányban láncolt listán viszont csak valamely elem elé tudunk beszúrni, ezért használunk egy lemaradó segédpointert. (Egy mozgó pointerrel is megoldható, de nem ezt a megoldást választottuk.)

Feladat ▶ Írjon szabványos C programot, amely a szabványos bemenetről olvassa be lakosok adatait: név, irányítószám, cím. A lakosok száma előre nem ismert. Írja ki a lakosok adatait irányítószám szerint növekvő sorrendben, azonos irányítószám esetén pedig cím szerinti ABC sorrendben (a program az angol ABC betűire működjön helyesen). Az irányítószám négy- vagy ötjegyű.

```

//*****
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//*****

//*****
typedef struct _lakos{
//*****
    unsigned irszam;
    char cim[150];
    char nev[60];
    struct _lakos * next;
} lakos;

//*****
void hiba(char t[]){
//*****
    printf("Hiba: %s\n",t);
    exit(1);
}

//*****
void beszur(lakos * start,const lakos * puj){
/* Az új elem azért pointer, mert így kevesebb adatot
kell mozgatni, nem kell eltárolni a struktúrát a
stack-en, csak a címét. A start-ról feltételezzük,
hogy nem NULL, mert van strázsa.
*/
//*****
    lakos * lemarad=start,*ptemp;
    start=start->next;
    // a start pointer a main-ben lévő start
    // másolata, megváltoztatható visszahatás nélkül

    // megkeressük, hová kell beszúrni

    while(start->next!=NULL //start nem a végstrázsan

```

```

    && (start->irszam < puj->irszam
        // irányítószám szerint növekvő sorrend
    || (start->irszam == puj->irszam
        // azonos irányítószám esetén...
        && strcmp(start->cim,puj->cim)<0))){
        // cím szerint növekvő sorrend
    lemarad=start;
    start=start->next;
}

// memóriafoglalás és beszúrás

ptemp=(lakos*)malloc(sizeof(lakos));
if(ptemp==NULL)hiba("Memoriafoglalás.");
*ptemp=*puj; // a teljes struktúra átmásolása,
            // a tömbökkel együtt!!!
ptemp->next=start;
lemarad->next=ptemp; // ez a két értékadás
                    // beszúrta lemarad és strat közé
}

//*****
void kiir(const lakos * start){
//*****
    for(start=start->next; start->next!=NULL;
        start=start->next)
        printf("Nev: %s\nCim: %u, %s\n\n",
            start->nev, start->irszam, start->cim);
}

//*****
void torol(lakos * start){
//*****
    lakos * lemarad=start->next;
    for(start=start->next->next; start->next!=NULL;
        lemarad=start, start=start->next)
        free(lemarad);
}

//*****
const unsigned MaxIrszam=100000;
//*****

//*****
int main(void){
//*****
    lakos strazsa2={MaxIrszam,"","",NULL}
    ,strazsa1={0,"","",&strazsa2};
    lakos temp={0,"","",NULL};
    int jovolt=1; // logikai érték, ha hiba a
                // beolvasásnál vagy vége, 0 lesz
    char s[100];

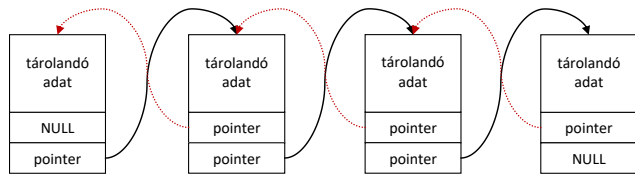
    do{
        printf("Irszam: ");
        if(fgets(s,100,stdin)==NULL)jovolt=0;
        if(jovolt && sscanf(s,"%u",&temp.irszam)!=1)
            jovolt=0;
        if(temp.irszam>MaxIrszam
            || temp.irszam<1000)
            jovolt=0;
        if(jovolt){
            printf("Cim: ");
            if(fgets(temp.cim,150,stdin)==NULL)
                jovolt=0;
        }
        if(jovolt){
            printf("Nev: ");
            if(fgets(temp.cim,150,stdin)==NULL)
                jovolt=0;
        }
        if(jovolt)beszur(&strazsa1,&temp);
    }while(jovolt);
    kiir(&strazsa1);
    torol(&strazsa1);
    return 0;
}

```

Érdekeség a két strázsa: nem dinamikus változók. A tényleges listát ezek közé szűrjük be. Start pointerre nincs szükség, mert a kezdőelem a *strazsa1*, ennek a *next* elemétől lépkedve eljuthatunk a lista végéig.

Minden függvény figyel arra, hogy a strázsa adatát ne használja, és ne próbálja felszabadítani.

24.3 Két irányban láncolt lista



Ha egynél több pointer van a struktúrában, akkor sokféle adatszerkezetet tudunk kialakítani. Egyik lehetőség a két irányban láncolt lista.

Előnyök:

- Mindkét irányban tudunk lépkedni az elemek között. Így például a gyorsrendező algoritmus használható lesz két irányban láncolt listán is, míg az egyirányún nem.
- Nincs szükség a lemaradó pointerre, ezáltal könnyebb a beszúrás vagy a csere.

Hátrány:

- Több pointert kell beállítani.
- Nagyobb helyfoglalás.

A következő példában nem bonyolítjuk a listát, ahogy az előző példában, egyszerűen egy egész szám lesz a listaelemekben. A listakezelő függvények sorát kiegészítjük egy cserélő és egy kereső függvénnyel.

```

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

//*****
typedef struct _szam{
//*****
    int n;
    struct _szam *prev,*next;
} szam;

//*****
void hiba(char t[]){
//*****
    printf("Hiba: %s\n",t);
    exit(1);
}

//*****
void beszur(szam * start,int n){
//*****
    szam * p=NULL;
    for(start=start->next;
        start->next!=NULL && start->n < n;
        start=start->next);
    p=(szam*)malloc(sizeof(szam));
    if(p==NULL)hiba("malloc sikertelen");
    // befűzés
    p->n=n;
    p->prev=start->prev;
    p->next=start;
    start->prev->next=p;
    start->prev=p;
}

//*****
void csere(szam * p1,szam * p2){
//*****
    // megcserél a listában két elemet
//*****
    if(p1==p2) return; // ha szomszédosak:
    if( p1->next==p2 || p1->prev==p2 ){
        if(p1->prev==p2){ // pointerek megcserélése
            szam *temp=p1;
            p1=p2;
            p2=temp;
        }
        p1->next=p2->next; // mindig rajzoljunk!
        // figyeljünk a sorrendre is!
        p2->prev=p1->prev;
        p2->next=p1;
        p1->prev=p2;
        p2->prev->next=p2;
        p1->next->prev=p1;
    }
}

```

```

else{ // nem szomszédos
// bonyolult => segédpointereket használunk
szam * szomszed1=p1->prev;
szam * szomszed2=p1->next;
szam * szomszed3=p2->prev;
szam * szomszed4=p2->next;
szomszed1->next=p2;
szomszed2->prev=p2;
szomszed3->next=p1;
szomszed4->prev=p1;
p2->prev=szomszed1;
p2->next=szomszed2;
p1->prev=szomszed3;
p1->next=szomszed4;
}
}

//*****
void torol(szam * start){
//*****
szam * p=start->next;
if(p->next==NULL) return;
for(p=p->next ;p->next!=NULL; p=p->next)
free(p->prev);
free(p->prev);
p->prev=start; // a két strázsa egymásra mutat,
// így korrekt a törlés
start->next=p;
}

//*****
void kiir(szam * start){
//*****
for(start=start->next; start->next!=NULL;
start=start->next)
printf("%dn",start->n);
}

//*****
szam * keres(szam * start,int n){
//*****
for(start=start->next; start->next!=NULL;
start=start->next)
if(start->n==n) return start;
return NULL;
}

//*****
int main(){
//*****
szam strazsal, strazsa2;
int n;
szam *p1=NULL, *p2=NULL;

// A két strázsa láncba fűzése
strazsal.prev=strazsa2.next=NULL;
strazsal.next=&strazsa2;
strazsa2.prev=&strazsal;

// beolvasás
while (scanf("%d", &n)==1 && n!=0)
beszur(&strazsal, n);

// függvények kipróbálása
p1=keres(&strazsal, 3);
p2=keres(&strazsal, 6);
if (p1!=NULL && p2!=NULL) cserel(p1, p2); //ha
// valamelyik szám nincs a listában, nem cserélhet
kiir(&strazsal);
torol(&strazsal);
return 0;
}

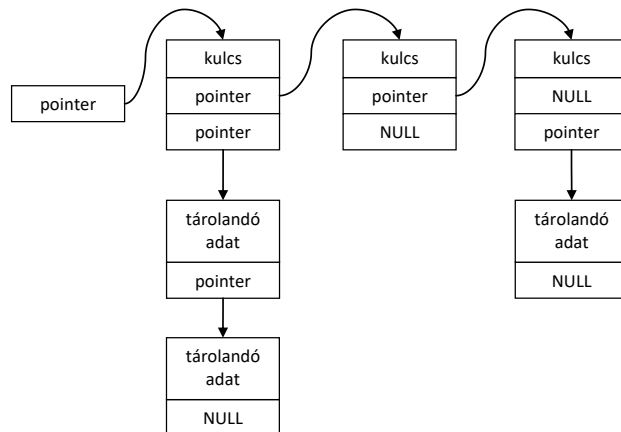
```

Ha magunk írunk listakezelést, nem kell, hogy kéznél tartunk valami mintakódot a beszúrás vagy a cserézés függvényhez. A szükséges pointer-másolásokat könnyedén kitalálhatjuk, ha rajzolunk, méghozzá folyamatosan: mindig rajzoljuk be a pointernek aktuális állapotát!

24.3 Fésűs lista

Itt az elemek már nem mind sorban egymás után következnek, azaz gyorsabban megtalálhatjuk a keresett elemet. Kétféle struktúra kell a tároláshoz:

- Amelyik a fésű szárát tárolja. Ennek tartalma: kulcs, pointer a következő szár elemre, pointer az első fog elemre.
- Amelyik a tényleges adatokat tárolja: közönséges láncolt lista elem.



A fésűs listához nem nézünk építő/bontó kódot, csak bejárást.

A listában telefonkönyvet tárolunk, a szárban a kulcselemek a vezetéknév kezdőbetűi, ABC sorrendben. A fogakban az adatok szintén ABC sorrendben vannak.

```

//*****
typedef struct _fogelem{
//*****
char nev[50];
char cim[150];
char telefonszam[30];
struct _fogelem * next;
} fogelem;

//*****
typedef struct _nyeielem{
//*****
char kulcs;
struct _nyeielem * pkovnyel;
fogelem * pfog;
} nyeielem;

//*****
fogelem * keresl(nyeielem * start, char keresett[]){
//*****
fogelem * pfog;
nyeielem * pnyel;
// addig megyünk, míg meg nem találjuk
// a keresett kezdőbetűt
for(pnyel=start; start->pkovnyel!=NULL
&& start->kulcs<keresett[0];
start=start->pkovnyel);
// ha nem találjuk, NULL-t adunk vissza
if(start->kulcs != keresett[0]) return NULL;
//innentől közönséges láncolt listán megyünk
for(pfog=start->pfog; pfog->next!=NULL &&
strcmp(pfog->nev, keresett)<0; pfog=pfog->next);
// vagy megtaláltuk, vagy nincs a listában
return strcmp(pfog->nev, keresett)==0 ? pfog
: NULL;
}

```

Gondolkozzunk együtt!

Oldja meg egyedül!

25. Állapotgép (véges automata)

Előfordulnak olyan feladatok, amikor nagyobb mennyiségű adatot kell feldolgoznunk, de nincs szükség az adatok tárolására, viszont a korábban beérkezett adatok befolyásolják, hogy az adott lépésben mit kell tennünk az aktuális adattal.

Ilyen például, amikor egy ismeretlen hosszúságú szöveg érkezik hozzánk, és ebből a szövegből valamit ki kell szűrünk, vagy valamit ki kell cserélnünk másvalamire.

A szövegfeldolgozásnál is jobban igényli az állapotgépes megvalósítást sok valós idejű algoritmus, például egy mérőműszert vezérlő szoftver.

- Feladat: a szoftver állítsa be a termosztát hőmérsékletét! Várja meg, míg stabilizálódik a hőmérséklet! Adjon a mérő eszközre egységugrás jelet, és másodpercenként n darab mérést végezve rögzítse a választ mindaddig, míg a kimeneten be nem áll a stabil állapot! Ezután ismételve meg a mérést más hőmérsékleten!

A fenti esetben két bizonytalan tényező is van: mennyi idő alatt áll be a hőmérséklet, és mennyi idő alatt áll be a válasz?

A következő állapotokat vehetjük fel:

- A – Kezdeti állapot, parancs kiadása új hőmérséklet beállítására (csak egy lépésnyit vagyunk ebben az állapotban)
- B – Figyeljük a hőmérsékletet, míg stabil nem lesz. Mindaddig ebben az állapotban maradunk, amíg a hőmérséklet változik. A stabilitást például úgy definiáljuk, hogy a legutolsó száz mérés eredményét eltároljuk, és ha nincs 0,1 Celsiusnál nagyobb eltérés a legalacsonyabb és legmagasabb hőmérséklet között, akkor stabilnak tekintjük
- C – Ráadja az egységugrás gerjesztést (csak egy lépésnyit vagyunk ebben az állapotban)
- D – Addig maradunk ebben az állapotban, míg a kimeneti jel nem stabilizálódik, közben fájlban rögzítjük a mérési eredményeket.

A fenti állapotgépet egy ütemező hívja, mondjuk másodpercenként tízszer, vagy amilyen sűrűn akarjuk mérni az értékeket. Több hőmérséklet esetén az egész állapotgép egy ciklus belsejében lesz, ami minden menetben más beállítandó hőmérsékletet és mérési eredményeket tároló fájlnevet biztosít.

25.1 Kommentyszűrő

Nézzünk konkrét példát állapotgépes feladatra!

Példa: Olvassunk be a szabványos bemenetről egy C programot, és írjuk úgy a szabványos kimenetre, hogy a benne található `/* */` megjegyzéseket töröljük! Egyszerűsítésként feltételezzük, hogy a programban lévő sztring konstansok nem tartalmaznak `/*` vagy `*/` párost. A szöveg végét EOF jelzi.

- Nem tudjuk, milyen hosszú egy sor
→ nem olvashatunk tömbbe egy sor
→ Karakterenként olvassunk!

Nem kell eltárolni az adatokat, rögtön ki lehet írni a beolvasott karaktert, ha biztosak vagyunk benne, hogy nem megjegyzés része.

1. próbálkozás ► Essünk neki, mint majom a farkának! Próbáljunk kitalálni valami megoldást!

Például elkezdünk gondolkodni:

- A karaktereket egyenként olvassuk egy while ciklusban.
- Oké, de ha `'/'` jött, akkor meg kell nézni, hogy a következő `'*'`-e! (Ha nem, írjuk ki a karaktert!)
- Ha `'*'`, akkor megjegyzésben vagyunk, tehát innentől egész a következő `'*'`-ig nem írjuk ki a karaktereket, azaz olvassunk a következő `*/`-ig! (És közben figyeljük, hogy jött-e EOF, mert hát ki tudja?)
- Ha `'*'` jött, akkor nézzük meg, hogy `'/'`-e a következő! Ha igen, akkor ez után megint ki kell írni a karaktereket. Ha

viszont nem `'/'` jött, akkor még mindig a megjegyzésben vagyunk. Akkor viszont nem kellett volna kijönni a ciklusból...

```
int main(void) {
    int ch; // a getchar int-et ad vissza
    while((ch=getchar())!=EOF){
        if(ch=='/'){
            ch=getchar();
            if(ch=='*'){
                while((ch=getchar())!=EOF && ch!='*');
                // ha EOF jött => vége
                // ha '/' jött, vége a megjegyzésnek
                // ha más jött, még mindig a
                // megjegyzésben vagyunk...
            }
            else if(ch!=EOF){
                putchar('/');
                putchar(ch);
            }
        }
        else putchar(ch);
    }
}
```

Itt most elakadtunk, de persze további bűvészkedéssel elérhetjük, hogy működjön a dolog.

2. próbálkozás ► További probléma, hogy egy csomó helyen olvastunk be, és ez így nehézkesé és áttekinthetetlené teszi a kódot. Megoldható, hogy csak egyszer kelljen beolvasni? Van-e valami szisztematikusabb módszer a megoldásra?

A megoldás természetesen az állapotgép.

- Az, hogy a következő beolvasott karakterrel mit kell tenni, attól függ, hogy a korábbi karakterek mik voltak. Kezdetben például ki kell írni a karaktereket, ha viszont most megjegyzésben vagyunk, akkor nem szabad kiírni a következő karaktert.
- Hozzunk létre egy változót, melynek segítségével azt tartjuk nyilván, hogy mit kell tennünk a következő karakterrel. A változó értéke jelzi, hogy normál kiíró helyzetben vagyunk, vagy esetleg megjegyzésben. Ez a változó lehetne mondjuk egy egész szám. Ha értéke 1, akkor normál állapotban vagyunk, ha pedig 2, akkor megjegyzésben.
- Jobban olvasható kódot kapunk, ha nem számokkal jelezzük a program állapotait, hanem konstansokkal, melyek neve jelzi az állapotot. Erre a célra tökéletes az **enum** típus!
- Vannak még további állapotaink is. Megjegyzésbe csak akkor kerülünk, ha egymást követi egy `/*` páros, önmagában egyik sem elég. Megjegyzésből kikerülni is csak `*/` párossal lehet. A `/` után tehát még egyet kell olvasnunk, de ezt úgy akarjuk megtenni, hogy a kódban csak egyszer szerepeljen a beolvasás. Legyen egy olyan állapotunk is, amelyik a `/` jel érkezése után `*/`-ot vár, és egy olyan, amelyik `*/` után `/`-t!

Ezek alapján felírunk egy **állapotábrát**:

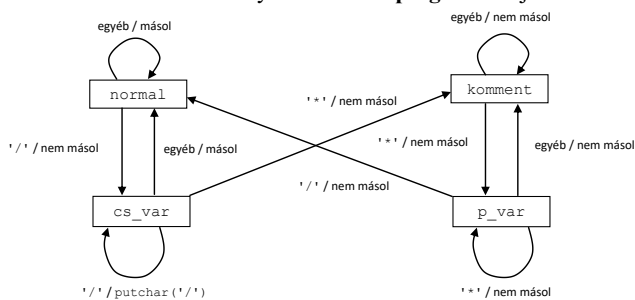
Állapot	'*'	'/'	egyéb
normál	normál	csillagra vár	normál
csillagra vár	megjegyzés	csillagra vár	normál
megjegyzés	perre vár	megjegyzés	megjegyzés
perre vár	perre vár	normál	megjegyzés

Írjuk fel emellé a **tevékenységtáblát** is!

Állapot	'*'	'/'	egyéb
normál	másol	nem másol	másol
csillagra vár	nem másol	előző <code>'/'</code> kiírása, nem másol	előző <code>'/'</code> kiírása, másol

megjegyzés	nem másol	nem másol	nem másol
perre vár	nem másol	nem másol	nem másol

A táblázatoknál látványosabb az **állapotgráf** felrajzolása:



Állapottábla alapján könnyebb a kódolás, és jobban látszik belőle, mely állapotok összevonhatók, továbbá sok állapot esetén a gráf követhetatlenné válik.

Az állapottábla+tevékenységtábla, vagy az állapotgráf alapján a program lekódolása egyszerű, mechanikusan végezhető feladat.

```
#include <stdio.h>
typedef enum {normal, komment, cs_var, p_var} állapotok;
// állapotgepes modellhez

int main(void) {
    int ch;
    állapotok állapot=normal;

    while((ch=getchar()) != EOF) {
        switch (állapot) {
            case normal:
                if(ch=='/') putchar(ch);
                else állapot=cs_var;
                break;
            case cs_var:
                if(ch=='*') állapot=komment;
                else {
                    putchar('/');
                    if(ch=='/'){
                        putchar(ch);
                        állapot=normal;
                    }
                }
                break;
            case komment:
                if(ch=='*') állapot=p_var;
                break;
            case p_var:
                if(ch=='/') állapot=normal;
                else if(ch!='*') állapot=komment;
                break;
        }
    }
}
```

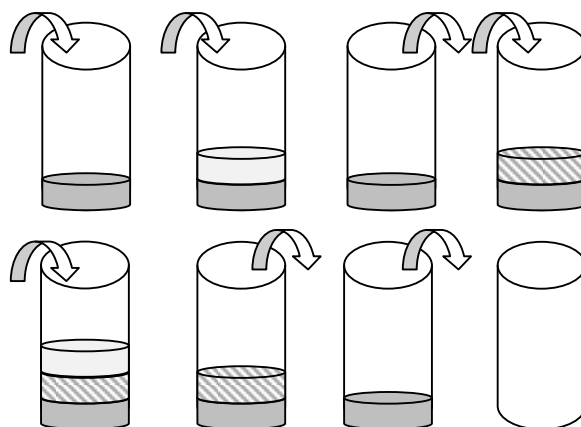
Gondolkozzunk együtt!

Oldja meg egyedül!

26. Rekurzió

26.1 A verem adatszerkezet

A **verem (stack)** olyan adatszerkezet, melyből az utoljára betett adatot tudjuk kivenni először. A „verem” kifejezés elég szemléletes: ha egy valódi verembe dolgokat dobálunk, akkor az újonnan bedobott legfelülre kerül, tehát ha ki akarunk venni valamit, akkor fordított sorrendben vehetjük ki a tárgyakat, mint ahogy bedobtuk.



A veremnek két utasítása van: bedob (push) és kivesz (pop).

A C nyelvben nincs verem típus, ha ilyenre van szükségünk, nekünk kell megvalósítani. Vermet létrehozhatunk például tömb vagy láncolt lista segítségével. Ha tömböt használunk, egyszerűbb a kód, viszont a verem fix helyet foglal, és ha betelik, akkor nem tudunk több elemet beletenni. Láncolt lista esetén a verem helyfoglalása a belepakolt adatok méretével arányos, és csak a számítógép memóriájának mérete korlátoz, cserébe valamivel bonyolultabb a hozzá tartozó kód.

Nézzünk egy nagyon egyszerű verem megvalósítást:

```
#include <stdio.h>

int t_verem[100], n=0;

void push(int uj) { t_verem[n++]=uj; }
int pop() { return t_verem[--n]; }

int main(void) {
    push(5);
    push(13);
    push(8);
    printf("%d\n", pop() ); // 8
    printf("%d\n", pop() ); // 13
    printf("%d\n", pop() ); // 5
    return 0;
}
```

A veremben egész értékeket tárolhatunk.

Egészítse ki a fenti programot veremtúlszordulás ill. alulszordulás elleni védelemmel, azaz ne engedje, hogy teli verembe írjunk ill. üres veremből elemet vegyünk ki! Alakítsa át úgy a programot, hogy az ne használjon globális változót!

Készítsen double értékek tárolására alkalmas vermet láncolt listával! Egyirányú listát használjon strázsa nélkül! Az új elemet a lista elejéhez fűzze!

26.2 Függvényhívás

A legtöbb számítógépen verem segítségével történik

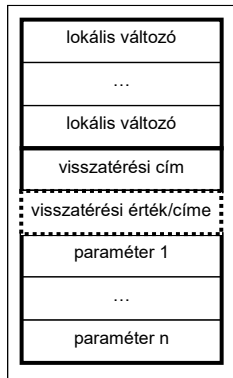
- a paraméterek átadása a függvényeknek.
- a visszatérési értékek visszaadása.
- a lokális változók tárolása.
- a függvényből való visszatérés címének tárolása.

Minden lefordított programnak van egy saját vereme, melyben tetszőleges típusú adatokat tárolhat. Visual C++-ban lefordított programok esetén 1 MB az alapértelmezett stack méret, melyet a Properties - Configuration Properties - Linker - System - Stack Reserve Size helyen módosíthatunk.

Készítsen `double` és `int` értékek közös tárolására alkalmas vermet! Ezt úgy valósítsa meg, hogy később könnyű legyen úgy kiegészíteni a kódot, hogy más típusú adatokat is el tudjunk tárolni!

Tipp: használjon `unsigned char` elemű tömböt a stack megvalósításához! Minden adattípusnak legyen saját `push/pop` függvénye, pl. `push_int`, `pop_double`! A `push` függvényben hozzon létre egy `unsigned char *` típusú pointert, mellyel mutasson a bemenő adatra, pl.: `(double be){unsigned char *p=(unsigned char *)&be;...}`, és a pointer által mutatott „tömbből” másolja a bájtokat a stack tömbjébe (a példában `sizeof(double)` darab bájt másolandó).

Egy függvényhez tartozó veremrészlet például a következőképpen nézhet ki:



A paramétereket a hívó teszi bele, méghozzá fordított sorrendben. Ez azért szükséges, mert ha változó paraméterszámú függvényt hívunk (pl. `printf`, `scanf`, stb.), a függvény az első paraméter alapján tudja, hogy hány és milyen típusú értékkel hívták meg.

A visszatérési érték/cím, és a szaggatott körvonala magyarázatra szorul: a paramétek és visszatérési értékek átadási módjáról a fordító dönt, ez sokféleképp működhet. A Visual C++ abban az esetben, ha a visszatérési érték elfér egy-két regiszterben, akkor nem a veremben adja vissza, hanem a processzor valamelyik regiszterében. Ha a visszatérési érték egy nagyobb struktúra, akkor a hívó függvény létrehoz egy ideiglenes lokális változót a visszatérési értéknek, és a verembe ennek címét helyezi el. Más fordító viszont dolgozhat úgy, hogy a visszatérési érték nem ideiglenes változóban van, hanem közvetlenül a veremben.

Példaprogram ► A következő példaprogram egy másodfokú polinom értékét számítja ki:

```
#include <stdio.h>

int masodfok(int a, int b, int c, int x){
    int res = a*x*x;
    res += b*x+c;
    return res;
}

int main(void){
    int y,x=5;
    y=masodfok(2, 6, 3, x);
    printf("%d\n", y);
    return 0;
}
```

Nézzük meg a lefordított gépi kódot!
Először a `main` egy részlete:

```
; 11 : y=masodfok(2, 6, 3, x);
00025 8b 45 ec mov eax,DWORD PTR _x$[ebp]
00028 50 push eax
00029 6a 03 push 3
0002b 6a 06 push 6
0002d 6a 02 push 2
0002f e8 00 00 00 00 call _masodfok
00034 83 c4 10 add esp,16 ; 00000010H
00037 89 45 f8 mov DWORD PTR _y$[ebp],eax
```

Az `x` változó az `eax` regiszterbe kerül, innen a `push` a program verembe helyezi (csak konstansokat vagy regiszter értékeket lehet a verembe tenni, változókat közvetlenül nem), ezután a három konstans érték következik: fordított sorrendben kerülnek a verembe, mint ahogy a paraméterlistában szerepelnek.

A `call` utasítás a verem tetejére helyezi az őt követő utasítás címét, azaz a visszatérési címet, és utána a függvényre ugrik.

Visszatérve a függvényből az `add esp,16` 16-ot ad a veremmutató regiszterhez, azaz „törli belőle” a négy paramétert, majd az `eax` regiszterben lévő visszatérési értéket `y`-ba másolja.

Lássuk a `masodfok` függvényt!

```
; COMDAT _masodfok
TEXT SEGMENT
_res$ = -8 ; size = 4
_a$ = 8 ; size = 4
_b$ = 12 ; size = 4
_c$ = 16 ; size = 4
_x$ = 20 ; size = 4
_masodfok PROC NEAR ; COMDAT

; 3 : int masodfok(int a, int b, int c, int x){
00000 55 push ebp
00001 8b ec mov ebp, esp
00003 81 ec cc 00 00 00 sub esp,204 ; 000000ccH
00009 53 push ebx
0000a 56 push esi
0000b 57 push edi
0000c 8d bd 34 ff ff ff lea edi,DWORD PTR[ebp-204]
00012 b9 33 00 00 00 mov ecx,51 ; 00000033H
00017 b8 cc cc cc cc mov eax,-858993460; ccccccccH
0001c f3 ab rep stosd

; 4 : int res = a*x*x;
0001e 8b 45 08 mov eax,DWORD PTR _a$[ebp]
00021 0f af 45 14 imul eax,DWORD PTR _x$[ebp]
00025 0f af 45 14 imul eax,DWORD PTR _x$[ebp]
00029 89 45 f8 mov DWORD PTR _res$[ebp],eax

; 5 : res += b*x+c;
0002c 8b 45 0c mov eax,DWORD PTR _b$[ebp]
0002f 0f af 45 14 imul eax,DWORD PTR _x$[ebp]
00033 03 45 10 add eax,DWORD PTR _c$[ebp]
00036 03 45 f8 add eax,DWORD PTR _res$[ebp]
00039 89 45 f8 mov DWORD PTR _res$[ebp],eax

; 6 : return res;
0003c 8b 45 f8 mov eax,DWORD PTR _res$[ebp]

; 7 : }
0003f 5f pop edi
00040 5e pop esi
00041 5b pop ebx
00042 8b e5 mov esp, ebp
00044 5d pop ebp
00045 c3 ret 0
_masodfok ENDP
_TEXTENDS
```

Az 1-essel jelzett részben konstansokat definiál az assembly, ebből nem lesz gépi kód, csak az assembly kód olvashatóságát segíti. A konstansok azt mondják meg, hogy a verem `ebp` munkaregiszterének értékéhez képest (melybe a 2-es rész elején az `esp`, azaz a verem tetejét jelző regiszter értéke kerül) hány bájtnyira található az adott lokális változó. (Az x86-os rendszerekben a verem lefelé nő.)

A 2-es részben a függvény által használt és megőrzendő regiszterek mentése történik. A függvény használja ezeken kívül az `eax`-et ill. az `ecx`-et, ezeket azonban nem kell tárolni, mert munkaregiszterek.

A 3. részben a matematikai műveletek végrehajtása történik.

A 4. részben először az `eax` regiszterbe másolódik a `res` változó (az `eax`-ben kerül visszaadásra a visszatérési érték), majd visszatöltődnek a 2-es részben elmentett regiszterek, végül a `ret` utasítás visszaugrik a hívás helyére, miközben „törli” a visszatérési címet a veremből az `esp` regiszter növelésével.

26.3 Rekurzió

Rekurzív az a függvény, amely közvetlenül vagy közvetve önmagát hívja.

Ha egy függvény saját magát hívja, azt **önrekurzió**nak nevezzük.

Ha a függvények egy csoportja kölcsönösen egymást hívja, azt **kölcsönös rekurzió**nak nevezzük. (Vagyis A függvény hívja B függvényt, B függvény pedig A -t, de ez kiterjeszhető C , D , E ... függvényekre is.)

Rekurzióval több olyan probléma megoldható egyszerűen, mely rekurzió nélkül csak nagy nehézségek árán. A bináris fákkal foglalkozó 26. fejezetben számos rekurzív függvénnyel találkozni fogunk.

A rekurzív függvényhívás gondolata nagyon hasonló a matematikából ismert teljes indukciós bizonyítási módszerhez: az állítást belátjuk 1 -re, majd $n-1$ alapján n -re. Az egyik legegyszerűbb rekurzív definíció a faktoriálisé: 1 faktoriálisa 1 , n faktoriálisa $(n-1)! * n$.

A rekurzív függvények természetesen nem a végtelenségig hívják saját magukat, mindig van egy leállási feltétel, azaz a rekurzív függvényeknek szükséges kelléke a feltételes elágazás.

Nézzük meg konkrétan a faktoriális példáját!

```
#include <stdio.h>

int fakt(int n){
    int v;
    v = (n<2) ? 1 : n*fakt(n-1);
    return v;
}

int main(void){
    printf("4 faktoriálisa=%d\n",fakt(4));
}
```

Itt a leállási feltétel az $n < 2$ IGAZ esete, mert ekkor a függvény 1 -et ad vissza. Ha $n \geq 2$, akkor $n * fakt(n-1)$, ami rekurzív függvényhívás.

Hogyan képzeljük el a fejünkben a rekurzív függvényhívást? Íme, a faktoriális példa kifejtett változata:

```
#include <stdio.h>

int fakt_1(int n_1){
    int v_1;
    v_1 = 1;
    return v_1;
}

int fakt_2(int n_2){
    int v_2;
    v_2 = (n_2<2) ? 1 : n_2*fakt_1(n_2-1);
    return v_2;
}

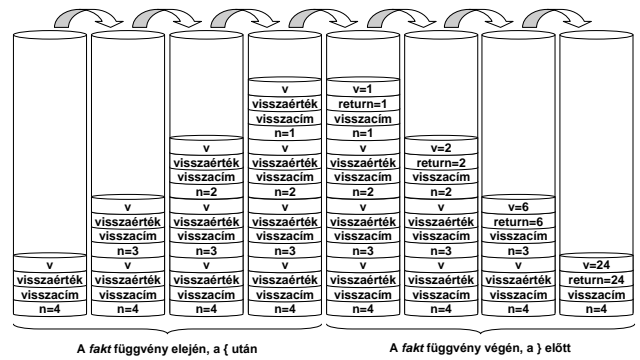
int fakt_3(int n_3){
    int v_3;
    v_3 = (n_3<2) ? 1 : n_3*fakt_2(n_3-1);
    return v_3;
}

int fakt_4(int n_4){
    int v_4;
    v_4 = (n_4<2) ? 1 : n_4*fakt_3(n_4-1);
    return v_4;
}

int main(void){
    printf("4 faktoriálisa=%d\n",fakt_4(4));
}
```

Kövesse végig fejben a program működését!

Látható, hogy ha a *fakt* függvény magát hívja, akkor nem íródik felül az n vagy a v változó, mert minden híváskor új keletkezett a veremben, valahogy így:



A faktoriális kiszámíthat ciklussal is, így:

```
int fakt(int n){
    int i,s=1;
    for(i=1; i<=n; i++) s*=i;
    return s;
}
```

A két megoldás nagyjából ugyanolyan hosszú, ugyanolyan bonyolult. Melyiket választjuk? Ha egy feladat ciklussal való megoldása hasonló bonyolultságú, mint a rekurzív megoldás, akkor a **ciklust érdemes választani**, mert az általában gyorsabb és kevesebb memóriát használ.



Írjuk ki ► egy tömb elemeit fordított sorrendben! Íme a rekurzív és a ciklusos megoldás:

```
void ford1(int t[],int n){
    if(n>0){
        ford1(t+1,n-1);
        printf("%d",t[0]);
    }
}

void ford2(int t[],int n){
    int i;
    for(i=n-1;i>=0;i--){
        printf("%d",t[i]);
    }
}
```

Ha egy 1000 elemű tömböt szeretnénk így kiírni, az első megoldás 32 bites rendszerben $1000 \times 12 = 12000$ bajtnyi helyet foglal a veremből, míg a második mindössze 16 bajtot. Vigyázni kell a rekurzió mélységére, mert nagy mélység esetén a verem betelik, és a program elszáll.



Fibonacci sorozat ► A Fibonacci sorozat első két eleme 1, a többi eleme pedig az öt megelőző két elem összege, azaz: 1, 1, 2, 3, 5, 8, 13, 21... Mivel a sorozatot eleve rekurzív definícióval adjuk meg, nyilvánvaló a rekurzív megoldás. Emellett megnézzük a ciklussal való megoldást is. A következő függvények a sorozat n . elemének értékét adják:

```
int fibo1(int n){
    if(n<3) return 1;
    return fibo1(n-2)+fibo1(n-1);
}

int fibo2(int n){
    int a, b, i;
    for(i=2,a=b=1; i<n; i++){
        int c=b;
        b+=a;
        a=c;
    }
    return b;
}
```

Próbáljuk ki mindkét függvényt pl. 30-cal meghívni! A rekurzív függvény több másodpercig fut, míg a ciklusos függvény

nagyon gyorsan befejeződik. A rekurzió mélysége ezúttal nem okoz problémát, mert a mélység mindössze 30. Akkor miért ilyen lassú a rekurzió? Mert alacsony n érték esetén is rengetegszer hívja önmagát, ugyanis ha végiggondoljuk a működést, az n . Fibinacci számot $1+1+1+1+1+\dots+1$ módon számítja ki, azaz a fibol függvény többször hívódik, mint amennyi az általa visszaadott érték. Tehát hiába adja magát egy feladat rekurzív megoldása, nem biztos, hogy az a legjobb.



Az eddigi példákban mindig adtunk megoldást ciklussal és rekurzióval egyaránt. Mindig létezik mindkét megoldás? Igen, azonban nem biztos, hogy hasonló bonyolultságú.

Minden ciklus megvalósítható rekurzióval. Az átalakítás nagyon egyszerű:

Ciklus függvény:
Amíg feltétel igaz
 Ciklusmag

VÉGE.

Rekurzív függvény:
Ha feltétel igaz
 Ciklusmag
 Rekurzív függvény

VÉGE.

Minden rekurzió megvalósítható ciklussal. Elméletben. A gyakorlatban azonban a ciklussal történő megvalósítás nagyon bonyolult lehet, sok segédváltozót igényelhet. Rekurzióval például nagyon egyszerű a fabejárás, lásd a következő fejezetben. Próbálja megvalósítani rekurzió nélkül bármelyiket!

Egy másik rekurzív algoritmus, ami rekurzió nélkül nagyon bonyolult, a felületkitöltés (flood fill): egy pixelekből álló képen egy adott színű, más színekkel határolt területet más színre szeretnénk festeni.

Algoritmus:

```
Flood fill függvény(x, y, cserélendő szín, új szín)
Ha pixel(x,y)!=cserélendő szín
    return
pixel(x,y)=új szín
Flood fill(x-1, y, cserélendő szín, új szín)
Flood fill(x+1, y, cserélendő szín, új szín)
Flood fill(x, y-1, cserélendő szín, új szín)
Flood fill(x, y+1, cserélendő szín, új szín)
```

VÉGE.

A cserélendő és az új szín nem lehet azonos.
Meg tudja oldani rekurzió nélkül?

Kiírás más számrendszerben ► Írjon függvényt, amely két pozitív egész számot kap paraméterként, és az elsőt a másodikkal megadott számrendszerben írja ki! A számrendszer 2-10 között lehet.

Meg kell határoznunk a szám számjegyeit, és ezeket egymás után kiírni.

Például ki szeretnénk írni a 835-öt 10-es számrendszerben. Az első számjegyet megkapjuk, ha elosztjuk 100-zal (egész osztás): $835/100=8$, majd levonjuk a $8*100$ -at, kapunk 35-öt. Ezt 10-zel osztjuk, stb. A probléma ezzel a megoldással, hogy tudnunk kell előre, hány jegyű a szám, mert ebből tudjuk meg, hogy 10 melyik hatványával kell osztanunk. Egy ciklussal meghatározható a kívánt hatvány. (A 00000835 stílusú eredményt adó „ez a hatvány biztosan elég lesz” típusú megoldás kerülendő, mert elvi hibás).

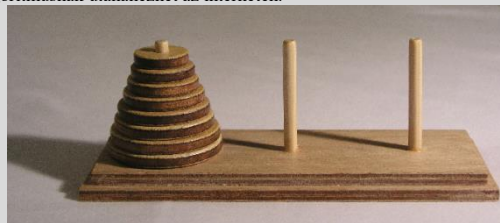
```
void szamrendszer(int x,int n){
int temp=x,oszt;
for(oszt=1; temp!=0; temp/=n, oszt*=n);
for(oszt/=n; x!=0; x%=oszt, oszt/=n)
printf("%d",x/oszt);
}
```

Elegánsabb megoldás adható rekurzióval. A következő módszer a szám számjegyeit fordított sorrendben adja, viszont nem kell tudnunk, hány jegyű a szám: $835\%10=5$, $835/10=83$, $83\%10=3$, $83/10=8$, $8\%10=8$, $8/10=0$. Fordított sorrendben nem írhatjuk ki a számjegyeket. Ha pl. láncolt listával eltároljuk a szám számjegyeit, akkor erősen túlbonyolítottuk a megoldást. Tömböt nem használhatunk, mert nem ismerjük előre a jegyek számát azaz a tömb méretét, fix méret megadása elvi hiba lenne. Rekurzív függvénnyel viszont könnyű megfordítani a sorrendet:

Rekurzív függvény (szám, számrendszer)
Rekurzív függvény(szám eleje, azaz szám/számrendszer)
Írd ki: utolsó számjegy, azaz szám%számrendszer
VÉGE.

```
void szamrendszer(int x,int n){
if(x<n)printf("%d",x);
else{
szamrendszer(x/n, n);
printf("%d", x%n);
}
}
```

A rekurziót gyakran a Hanoi tornyai játékon keresztül szokták bemutatni. A történet úgy szól, hogy egy rúdon 64 db egyre csökkenő méretű korong található, melyeket Brahma szerzeteseinek át kell raknia a mellette lévő rúdra úgy, hogy naponta egy korongot mozgathatnak, és sosem kerülhet kisebb korongra nagyobb. Segítségül egy harmadik oszlop is felhasználható. Feladat: Írjon C függvényt, amely kiírja az A oszlopról B oszlapra történő pakolás lépéseit n korong esetén! (64-gyel nem érdemes próbálkozni, mert a lépések száma 2^n-1 .) Az algoritmusnak utánanézhet az interneten.



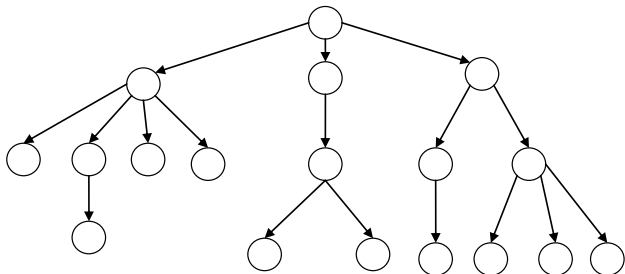
Ez a kép a Wikimedia Commonsból származik.

Gondolkozzunk együtt!

Oldja meg egyedül!

27. Dinamikus önhivatkozó adat-szerkezetek 2. - fák

A fa olyan gráf, melyben nincs hurok, azaz a gráf minden **csomópontjához** (node) csak egy úton lehet eljutni. Az adatszerkezetként használt fák irányított gráfok, azaz csak fentről lefelé lehet haladni.



Bár ez az alakzat ránézésre inkább gyökérre hasonlít. Ha feje tetejére állítanánk, akkor lenne belőle fa. Nem a fa az egyetlen, ami akkor mutatja meg valódi arcát, ha a feje tetejére állítjuk. Az alábbi *A zöldségekertész* című képet *Giuseppe Arcimboldo* festette.



A fa tetején lévő kiinduló elemet **gyökér elemnek** nevezzük.

Két csomópont **távolsága** azt jelenti, hogy hány másik csomóponton keresztül lehet eljutni hozzá. Mivel irányított gráfról van szó, nem lehet bármely csomópontból bármelyikbe eljutni.

Egy elem azon a **szinten** van, amennyi a gyökértől való távolsága. (A gyökér tehát a 0. szinten van.) A fenti ábrán az egy szinten lévő elemeket valóban egy szintre rajzoltuk, de ez nem kötelező.

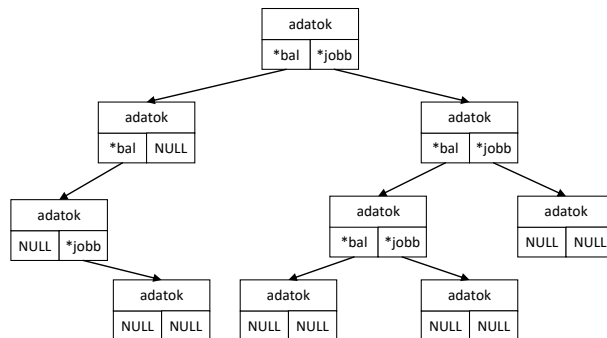
Azokat az elemeket, melyek között egy szint a különbség és kapcsolatban vannak egymással, **szülőnek** illetve **gyermeknek** nevezzük. Minden gyermeknek egy szülője van, egy szülőnek bármennyi gyermeke lehet. A fa **levele** olyan elem, melynek nincs gyermeke.

A fák rekurzív felépítésűek, mert bármely csomópontot kiválasztva azt tekinthetjük egy fa gyökerének. Fában keresni pl. úgy lehet rekurzívan, hogy megnézzük a gyökér elemet: ezt keressük-e. Ha nem, akkor minden gyermeket gyökérnek tekintve megismételjük az eljárást, míg van gyermek vagy meg nem találtuk a keresett elemet.

27.1 Bináris fa

Egy **fa** akkor **bináris**, azaz kétágú, ha egy szülőnek legfeljebb két gyermeke lehet. Az elnevezésnek tehát semmi köze a bináris számokhoz.

A bináris fa az informatikában önhivatkozó adatszerkezet. A fa csomópontjai struktúrák, melyek a csomópontban tárolt adatok mellett két pointert tartalmaznak: a bal illetve a jobb oldali gyermek címét. Ha nincs gyermek az adott irányban, a pointer értéke NULL.



Morse kód ► A következő példaprogram felépít egy Morse kódokat tároló bináris fát, majd dekódol egy Morse kóddal megadott szöveget.

A Morse kód megadása BNF leírással:

```
< sor > ::= " " < kód > "\n"
< kód > ::= < egyikód > | < kód > " " < egyikód >
< egyikód > ::= < karakter > < morse >
< morse > ::= < alap > | < morse > < alap >
< alap > ::= "." | "-"
```

Például: A . _ E . I . . M _ O _ _ P . _ . S . . . T _ Z _ . .

A Morse kód után szóköz van, a sort lezáró soremelés előtt szintén van szóköz!

A visszafejtendő szöveg formátuma:

```
< sor > ::= " " < kód > "\n"
< kód > ::= < morse > | < kód > " " < morse > | < kód > " "
< morse > ::= < alap > | < morse > < alap >
< alap > ::= "." | "-"
```

Például: . . . _ _ _ _ . _

A Morse karaktereket szóköz választja el egymástól. Ha egy-nél több szóköz van két karakter között, akkor ezek a kimeneten is megjelennek, mint szavakat elválasztó szóközök. A sort lezáró új sor karakter előtt kell legyen szóköz.

```

//*****
#include <stdio.h>
#include <stdlib.h>
//*****

//*****
typedef struct dat{
//*****
    char betu;
    struct dat *pont,*vonal;
}Morse,*MP;

//*****
void hiba(char t[]){
//*****
    fprintf(stderr,"Hiba: %s\n",t);
    exit(1);
}

//*****
void delfa(MP gyoker){
//*****
    if(gyoker==NULL) return;
    delfa(gyoker->pont);
    delfa(gyoker->vonal);
    free(gyoker);
}

```

```

//*****
MP feltolt(MP gyoker) {
//*****
    int c,aktbetu=0;
    MP futo;

    delfa(gyoker);
    gyoker=(MP)malloc(sizeof(Morse));
    if(gyoker==NULL)hiba("Elfogyott a memoria");
    gyoker->pont=gyoker->vonal=0;
    gyoker->betu=0;

    futo=gyoker;

    printf("Kerem a definiciot!\n");
    while((c=toupper(getchar()))!=EOF && c!='\n'){
        switch(c){
            case '.':{
                if(futo->pont==NULL){
                    futo->pont=(MP)malloc(sizeof(Morse));
                    if(futo->pont==NULL)
                        hiba("Elfogyott a memoria");
                    futo=futo->pont;
                    futo->betu=0;
                    futo->pont=futo->vonal=NULL;
                }
                else futo=futo->pont;
                break;
            }
            case '_':{
                if(futo->vonal==NULL){
                    futo->vonal=(MP)malloc(sizeof(Morse));
                    if(futo->vonal==NULL)
                        hiba("Elfogyott a memoria");
                    futo=futo->vonal;
                    futo->betu=0;
                    futo->pont=futo->vonal=NULL;
                }
                else futo=futo->vonal;
                break;
            }
            default:{
                futo->betu=aktbetu;
                if(isalnum(c))aktbetu=c;
                else aktbetu=0;
                futo=gyoker;
            }
        }
    }
    return gyoker;
}

//*****
void keres(MP gyoker) {
//*****
    int c;
    MP futo=gyoker;
    int jo=1;

    printf("Kerem a megfejtendo kodot!\n");
    while((c=toupper(getchar()))!=EOF && c!='\n'){
        switch(c){
            case '.':{
                if(jo){
                    if(futo->pont!=NULL)futo=futo->pont;
                    else jo=0;
                }
                break;
            }
            case '_':{
                if(jo){
                    if(futo->vonal!=NULL)futo=futo->vonal;
                    else jo=0;
                }
                break;
            }
            default:{
                if(futo->betu&&jo)putchar(futo->betu);
                else if(c==' ')putchar(' ');
                else putchar('?');
                futo=gyoker;
                jo=1;
            }
        }
    }
}

```

```

//*****
int main(void) {
//*****
    MP gyoker=NULL;
    gyoker=feltolt(gyoker);
    keres(gyoker);
    delfa(gyoker);
    return 0;
}

```

Az elem adata az a karakter, melyet a gyökérből az útvonalnak megfelelő pont-vonal kombináció ad. A bal és jobb ponttereket ezúttal *pont*-nak illetve *vonal*-nak nevezzük.

A *delfa* függvény rekurzív felépítésű: Először törli a gyermekeit (azok is az ő gyermekeiket, stb.), majd saját magát, így a fa minden eleme törlődik. A rekurzió leállási feltétele, hogy a kapott gyökér NULL pointer-e. Ez egy kellemes feltétel, mert így nyugodtan meghívhatja magát a függvény a jobb és bal részfára, nem kell vizsgálnia, hogy ott NULL pointer van-e.

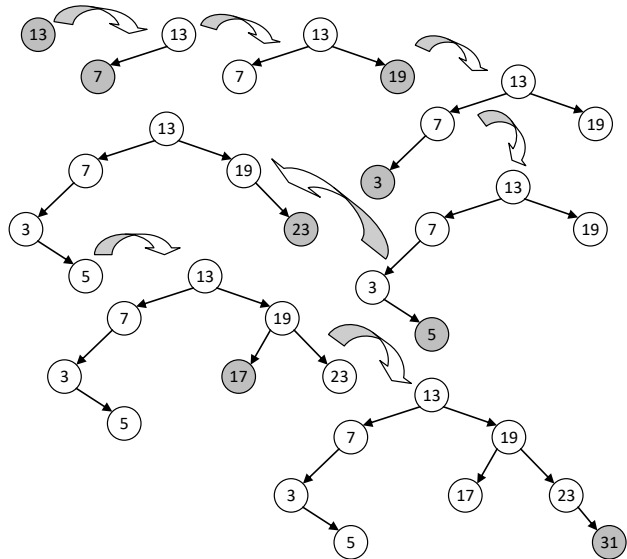
A fa feltöltését végző *feltolt* ill. a keresést végző *keres* függvény viszont ciklust használ. Meg tudná oldani rekurzióval is?

27.2 Rendezett fa – rendezőfa, kiegyensúlyozottság és kiegyensúlyozatlanság

A bináris fát leggyakrabban úgy építjük fel, hogy abban az elemek valamilyen szempont szerint rendezve legyenek.

Például egy egész elemeket tartalmazó fába a folyamatosan érkező elemeket úgy építjük be, hogy a gyökérelmentől balra csak nála kisebb, tőle jobbra csak nagyobb értékek kerüljenek, és ez az állítás bármely elemre igaz: tőle balra csak kisebbek, tőle jobbra csak nagyobbak vannak.

Legyen a bemenő számsor: 13, 7, 19, 3, 5, 23, 17, 31. Az ezekből felépülő rendezőfa a következő:



- Az első elem (13) lesz a fa gyökere.
- A 7 tőle balra kerül, mert kisebb nála.
- A 19 a gyökértől jobbra kerül, mert nagyobb nála.
- A 3 a gyökértől balra kerül, mert kisebb nála. Ott már van elem, ezért további szintet kell létrehozni a 3 számára. A 3 kisebb az 5-nél, ezért tőle balra kerül.
- Az 5 a gyökértől balra kerül, mert kisebb nála, a 7-től balra kerül, mert kisebb nála. A 3-tól jobbra kerül, mert nagyobb nála.
- ...

Vegyük észre, hogy **amennyiben ugyanezek a számok más sorrendben érkeztek volna, egészen más fát kaptunk volna!**



Miért jó a rendezett fa? Az inorder bejárásnál meglátjuk.

Milyen fát kaptunk volna, ha az elemek növekvő sorrendben érkeznek? 3, 5, 7, 13, 17, 19,...

Gyökér: 3 → töle jobbra 5 → töle jobbra 7 → töle jobbra 13... Ha felrajzoljuk, látjuk, hogy ez gyakorlatilag egy egyszerűen láncolt lista.

Ha az eredeti fában akarunk eljutni a legtávolabbi elemig, akkor a gyökértől hármat kell lépni. A láncolt lista-szerűen kapcsolódó bináris fa esetében viszont a legtávolabbi elem hét lépésre van a gyökértől. Nyilvánvalóan az a hatékonyabb, ha kevesebb lépésben lehet eljutni a legtávolabbi elemhez, ez pedig **kiegyensúlyozott fával** érhető el. A kiegyensúlyozottság azt jelenti, hogy bármely elemtől balra és jobbra is nagyjából ugyanannyi elem legyen. A láncolt lista-szerű felépítés szélsőségesen kiegyensúlyozatlan eset. Az elemek átrendezésével utólag ki lehet egyensúlyozni a fát, ennek megvalósításával azonban nem foglalkozunk.

27.3 A fabejárás típusai

Aszerint, hogy a fa gyökérelemének tartalmát mikor dolgozzuk fel a két ág feldolgozásához képest, háromféle bejárástípust különböztetünk meg:

Preorder bejárás ►

Preorder fabejáró függvény
elem feldolgozása
egyik ág bejárása
másik ág bejárása

Ezt a bejárást például akkor használjuk, ha a fa elemeit fájlba akarjuk menteni, majd onnan visszaolvasva az eredeti fát akarjuk visszakapni. A fenti fát 13, 7, 3, 5, 19, 17, 23, 31 sorrendben írja ki. Ez ugyan nem egyezik a számok eredeti sorrendjével, de láthatjuk, hogy ha ebből építünk rendezőfát, ugyanazt kapjuk vissza.

Inorder bejárás ►

Inorder fabejáró függvény
egyik ág bejárása
elem feldolgozása
másik ág bejárása

A fenti fában, ha egyik ág=bal, másik ág=jobb, akkor 3, 5, 7, 13, 17, 19, 23, 31 sorrendben kapjuk az elemeket, azaz növekvő sorrendben. Ha egyik ág=jobb, másik ág=bal, akkor pedig ellenkező, azaz csökkenő sorrendben. Így kell eljárni tehát, ha az elemekre sorrendben van szükség.

Postorder bejárás ►

Postorder fabejáró függvény
egyik ág bejárása
másik ág bejárása
elem feldolgozása

Ez a bejárás használatos a fa törlésekor.

Előfordulhat, hogy olyan megoldást alkalmazunk, amelyet nem lehet egyértelműen besorolni a három változat közé.

Rekurzió és bejárás ► A rekurzió kapcsán megemlítettük, hogy alkalmazását mindig meg kell fontolni:

- Előnyösebb-e a ciklussal való megoldás a rekurzívnál?
- Nem túl mély-e a rekurzió, és ezáltal nem telik-e be a verem?

Bináris fákról azt mondhatjuk, hogy ha a fa kiegyensúlyozott, akkor egyértelműen érdemes használni a rekurziót, mert jóval

egyszerűbb kódot eredményez, mint a ciklussal való megvalósítás. (Próbáljuk meg kiírni egy rendezett fából az elemeket növekvő sorrendben ciklus segítségével! És rekurzívan?) Rekurzió esetén a rekurzió mélysége megegyezik a fa mélységével, azaz az emeletek számával. Kiegyensúlyozott fa esetén ez az elemszám kettes alapú logaritmusával arányos, vagyis nem nagy. (0. szint: 1 elem, 1. szint: 2 elem, 2. szint: 4 elem, 3. szint: 8 elem...). A sebessége pedig a ciklusnak sem nagyobb.

Láncolt lista is bejárható rekurzívan, de ott ez a megoldás kerülendő, mivel a rekurziómélység megegyezik az elemszámmal.

27.4 Több ágú fák

Kis módosítással a kétágú fára alkalmazott algoritmusok és adatszerkezetek alkalmasak több ágú fák kezelésére is.

Pl.: 3 ágú fa:

```
typedef struct _elem3{
    int adat;
    struct _elem3 *bal,*kozep,*jobb;
}elem3;
```

Pl. n ágú fa:

```
typedef struct _elemn{
    int adat;
    unsigned n;
    struct _elemn **t; // n elemű din. pointertömb
}elemn;
```

Ezek bejárása ugyanúgy történhet rekurzív függvénnyel, mint a bináris fák.

```
Pl.
//*****
int osszead(elem3 * gyoker){
//*****
    if(gyoker==NULL) return 0;
    return osszead(gyoker->bal)
        +osszead(gyoker->kozep)
        +osszead(gyoker->jobb);
}
```

ill.

```
//*****
int osszead(elemn * gyoker){
//*****
    int sum=0,i;
    if(gyoker==NULL) return 0;
    for(i=0; i<gyoker->n; i++)
        sum+=osszead(gyoker->t[i]);
    return sum;
}
```

27.5 Két példa

1. példa ► Adottak a következő definíciók:

```
typedef struct bf {
    double ertek;
    struct bf *bmut, *jmut;
} bifa, *pbifa;
```

Készítsen egy olyan szabványos ANSI C függvényt, amely meghatározza az ilyen elemekből álló bináris fában tárolt értékek összegét. Az adatszerkezet "végét" a megfelelő mutató NULL értéke jelzi. A függvény bemenő paramétere a fa gyökerére mutató pointer, visszatérési értéke a fenti módon meghatározott érték legyen! Válasszon megfelelő paramétereket! Törekedjen gyors megoldásra!

Egy lehetséges megoldás:

```
double osszegez(pbif a gyoker){
    return gyoker!=NULL ? gyoker->ertek+
        osszegez(gyoker->bmut)+
        osszegez(gyoker->jmut)
        : 0.0;
}
```

2. példa ► Adott a következő definíció:

```
typedef struct mf {
    double adat;
    unsigned m;
    struct mf **p;
} magufa, *pmagufa;
```

Készítsen egy olyan C függvényt, amely meghatározza, egy ilyen *magufa* típusú elemekből álló fa *n*-edik szintjén található *adat*-ok összegét! Egy csomópont *p* mezője egy *m* elemű dinamikus tömbre mutat, amelyben a csomópont gyerekeire mutató pointereket tároljuk. Ha egy elemnek nincs gyereke, *p* NULL pointer. A függvény bemenő paramétere a fa gyökerére mutató pointer, valamint *n*, azaz a keresett szint. Visszatérési értéke az *n*-edik szinten tárolt adatok összege legyen! Nem használhat globális változót!

Egy lehetséges megoldás:

```
double adatosszeg(pmagufa p,int n){
    double s=0.0;
    unsigned i;
    if(!p) return 0;
    if(n==0) return p->adat;
    for(i=0; i<p->m; i++)
        s+=adatosszeg(p->p[i],n-1);
    return s;
}
```

Gondolkozzunk együtt!

Oldja meg egyedül!

28. Függvénypointer

A függvények kódja a memóriában található, csakúgy, mint a változók, tehát meg lehet mondani azt a memóriacímet, ahol az adott függvény kezdődik. A C nyelvben definiálhatók olyan pointerok, melyek függvények címeit tárolják. Ezeket a pointerket hasonlóan kezelhetjük, mint az adatpointereket, például tömböt szervezhetünk belőlük vagy átadhatjuk függvénynek paraméterként.

A következő példában létrehozunk egy olyan tömböt, melynek elemei *double* visszatérési értékű, egy *double* paraméterrel rendelkező függvények címei. Ezután feltöltjük a tömböt négy függvényt, melyek a *math.h*-ban találhatóak. Természetesen saját függvényeket is használhatnánk. Az utolsó sorban meghívjuk a tömb *n-1*-edik elemét.

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

int main(void){
    double (*t[4]) (double),d;
    int n;

    t[0]=sin;
    t[1]=cos;
    t[2]=exp;
    t[3]=tan;

    printf("Kerek egy valos szamot: ");
    scanf("%lg",&d);
    printf("Mit szamoljak?\n1. sin\n");
    printf("2. cos\n3. exp\n4. tan\n");
    scanf("%d",&n);

    if(n<1 || n>4) return;

    printf("Eredmeny: %g\n",t[n-1](d));

    return 0;
}
```

Egyszerű menüvezérelt program ► Megkérdezzük a felhasználotól, hogy összeadni, kivonni, szorozni vagy osztani szeretne-e, és ennek megfelelően választunk függvényt. A program könnyen bővíthető további függvényekkel. Ha a felhasználó rossz számot ír be, a program ismét kiírja a menüt. Ha nem számot ír be (hiba-kezelés), vagy a 0 menüpontot választja, a program kilép.

```
#include <stdio.h>

void osszead(double a,double b){
    printf("Osszeg: %g\n",a+b);
}

void kivon(double a,double b){
    printf("Kulonbseg: %g\n",a-b);
}

void szoroz(double a,double b){
    printf("Szorzat: %g\n",a*b);
}

void menukiir(void){
    printf("\n0 Kilep\n");
    printf("1 Osszead\n");
    printf("2 Kivon\n");
    printf("3 Szoroz\n");
    printf("\n");
}

int main(void){
    void (*t[3]) (double,double)={osszead,kivon,szoroz};
    int n;
    double a,b;

    menukiir();
    while(scanf("%d",&n)==1 && n!=0){
        if(n>0 && n<4){
            printf("Kerek ket szamot ,-vel elvalasztva: ");
            scanf("%lg,%lg",&a,&b);
            t[n-1](a,b);
        }
    }
}
```

```

    }
    menukiir();
}

return 0;
}

```

Integrálszámítás téglalap formulával ► A következő példa azt mutatja be, hogyan írhatunk függvényt, amely tetszőleges, egyváltozós (valós) függvény integrálját számítja ki téglalap formulával. A *teglalap* függvény *fv* függvény integrálját számítja ki $[a,b]$ intervallumon, az intervallumot n egyenlő részre osztja.

```

#include <stdio.h>
#include <math.h>

double teglalap(double a, double b, int n, double (*fv)(double)) {
    double dx=(b-a)/n, sum=0;
    int i;
    for(i=1; i<n; i++) sum+=fv(a+dx*i);
    return (sum+0.5*(fv(a)+fv(b)))*dx;
}

double x2(double x) {
    return x*x;
}

int main(void) {
    printf("I(sin[0,1])=%g\n", teglalap(0,1,100,sin));
    printf("I(x2[0,1])=%g\n", teglalap(0,1,100,x2));
    return 0;
}

```

A *teglalap* függvény negyedik paramétere tehát egy olyan függvény címe, amelynek mind a paramétere, mind a visszatérési értéke *double* típusú. A paraméterként átadott függvényt úgy hívjuk meg, mintha egyszerű függvény volna, nem pointer.

A függvénynek átadott függvénypointer használatát korábban láthattuk a *qsort* esetében. A *qsort* paraméterként kapta az összehasonlító függvényt.

A $fv(a+dx*i)$ helyett $(*fv)(a+dx*i)$ alakban is használható a függvénypointer, a két megoldás ekvivalens egymással. A pointeres írásmód a korábbi és a későbbi példákban is használható, de a függvény írásmód egyszerűbb, ezért ennél maradunk.

Integrálszámítás adaptív finomítással ► Az alábbi példa függvénypointert és rekurziót egyaránt tartalmaz.

```

#include <stdio.h>
#include <math.h>

double teglalap(double a, double b, int n, double (*fv)(double)) {
    double dx=(b-a)/n, sum=0;
    int i;
    for(i=1; i<n; i++) sum+=fv(a+dx*i);
    return (sum+0.5*(fv(a)+fv(b)))*dx;
}

double finomitas(double a, double b, double hiba, double (*fv)(double), double eloazo, int szint) {
    double kozep=(a+b)*0.5;
    double bal=teglalap(a, kozep, 16, fv);
    double jobb=teglalap(kozep, b, 16, fv);
    double uj=bal+jobb;

    if(szint>20) return uj; // ha túl nagy lenne a
    // rekurzió mélysége, nem megyünk tovább
    if(fabs((uj-eloazo)/uj)<hiba) return uj; // ha kész
    hiba*=0.5;
    return finomitas(a, kozep, hiba, fv, bal, szint+1)
    +finomitas(kozep, b, hiba, fv, jobb, szint+1);
}

double integral(double a, double b, double hiba, double (*fv)(double)) {
    return finomitas(a, b, hiba, fv,
    , teglalap(a, b, 16, fv), 1);
}

```

```

double x2(double x) {
    return x*x;
}

int main(void) {
    printf("I(sin[0,1])=%g\n", integral(0,1,1e-8,sin));
    printf("I(x2[0,1])=%g\n", integral(0,1,1e-8,x2));
    return 0;
}

```

Az *integral* függvény számítja ki *fv* függvény integrálját $[a,b]$ intervallumon. Az integrálásnál használt x irányú felosztás ebben az esetben nem egyenletes, mint a korábbi, *teglalap* formulát használó esetben, hanem ahol a függvény hullámosabb, ott sűrűbb a felosztás, ahol pedig egyenletesebb, ott ritkább.

Ezt úgy érjük el, hogy első lépésben kiszámítjuk az integrál közelítő összegét kevés (16) pontra a *teglalap* formulával (még az *integral* függvényben), majd (már a *finomitas* függvényben) vesszük az intervallum alsó és felső felét, és kiszámítjuk azok integrálját is, ugyancsak 16-16 osztóponttal. A két félre kapott integrál összege egy pontosabb közelítést ad, mert kétszer annyi osztópontot használtunk.

Megnézzük, hogy az új és az előző integrálközelítő összeg mennyire tér el egymástól. Ha az eltérés a hiba változóban magadottnál kisebb, akkor nem számolunk tovább. Ha nagyobb, akkor a két oldalon külön-külön számoltatunk egy pontosabb eredményt. Ha az egyik oldalon a következő lépésben elég pontos eredményt kapunk, a másik oldalon viszont nem, akkor a két oldalon eltérő felbontáshoz fogunk jutni.

A rekurzív hívás tehát megáll, ha elértük a kívánt pontosságot. Előfordulhat, hogy ezt sosem tudjuk elérni (számítási hibák miatt), ezért bevezettünk egy korlátozást: ha 20-nál nagyobb mélységre lenne szükség, akkor is leáll a folyamat.

Gondolkozzunk együtt!

Oldja meg egyedül!

29. Keresés tömbben

Két eset lehetséges: ha a tömb rendezve van a keresett adat szerint, és ha rendezetlen. Ha rendezetlen, akkor csak egyet tehetünk: elindulunk a tömb elejéről (vagy végéről) és minden elemet megvizsgálunk, hogy az-e a keresett. Ennek a triviális módszernek is van tudományos neve: **lineáris keresésnek** nevezzük.

Lineáris keresés ► Keressünk meg egy double tömbben egy értéket, és adjuk vissza az indexét! Ha nincs a tömbben, -1-et adjunk vissza!

```
int linkeres(double t[],int db,double mit){
    int i;
    for(i=0;i<db;i++)
        if(t[i]==mit)
            return i;
    return -1;
}
```

Bináris keresés ► Abban az esetben, ha a tömb rendezett a keresési kulcs szerint, a lineáris keresésnél gyorsabban is megtalálhatjuk a kívánt elemet. A leghatékonyabb keresési módszer a **bináris** vagy más néven **logaritmikus keresés**, melyet a következő példa szemléltet.

Keressük 19-et az alábbi tömbben:

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

2 3 5 7 11 13 17 19 23 29 31 37
 ↑ min ↑ max

2 3 5 7 11 13 17 19 23 29 31 37
 ↑ min ↑ med ↑ max

2 3 5 7 11 13 17 19 23 29 31 37
 ↑ min=med ↑ max

2 3 5 7 11 13 17 19 23 29 31 37
 ↑ min ↑ med ↑ max

2 3 5 7 11 13 17 19 23 29 31 37
 ↑ min ↑ max=med

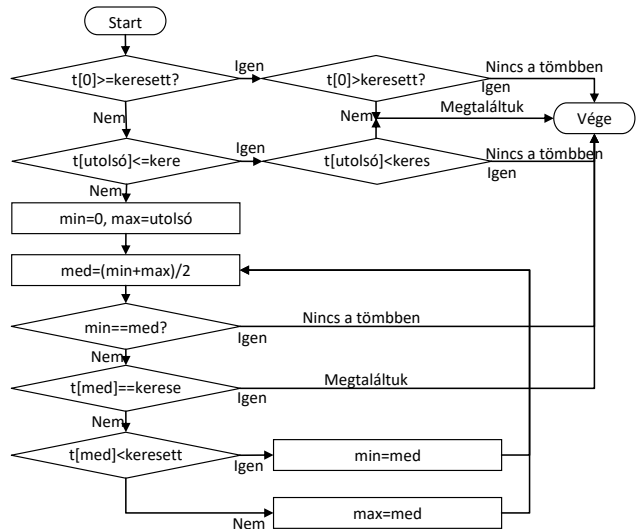
2 3 5 7 11 13 17 19 23 29 31 37
 ↑ min ↑ med ↑ max

2 3 5 7 11 13 17 19 23 29 31 37
 ↑ min=med ↑ max

2 3 5 7 11 13 17 19 23 29 31 37
 ↑ min ↑ med ↑ max

Azaz: a min és max indexek kezdetben a tömb két szélén állnak, majd minden lépésben feleződik köztük a távolság, körülkerítik a keresett elemet. A keresés ideje $\log_2 n$ nagyságrendű, ahol n a tömb számossága. (Lineáris keresésnél $n/2$ nagyságrendű).

A logaritmikus (bináris) keresés folyamatábrája a következő:



A feladat ugyanaz, mint a lineáris keresésnél. A fenti algoritmus megoldásítva:

```
int binkeres(double t[],int db,double mit){
    int min=0,max=db-1,med=(min+max)/2;

    if(t[min]>mit) return -1;
    if(t[min]==mit) return min;
    if(t[max]<mit) return -1;
    if(t[max]==mit) return max;

    while(med!=min && t[med]!=mit){
        if(t[med]<mit) min=med;
        else max=med;
        med=(min+max)/2;
    }
    return t[med]==mit ? med : -1;
}
```

Keresési idő: $O(\log_2 n)$

Bináris keresést valósít meg a *bsearch* szabványos könyvtári függvény, mely az *stdlib.h*-ban található.

```
void bsearch(
    void *key,
    void *base,
    size_t nelem,
    size_t width,
    int (*fcmp)(const void *elem1,const void *elem2)
);
```

key: a keresett elemet tároló változó címe
base: a tömb címe
nelem: elemszám
width: egy elem mérete
fcmp: két tömbelemet összehasonlító függvény címe

Az összehasonlító függvény ugyanolyan, mint amelyet a *qsort* is használ, lásd a 30.7 fejezetben.

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

int hasonlit(const void *p1, const void *p2){
    double *d1=(double*)p1;
    double *d2=(double*)p2;
    if(*d1<*d2) return -1;
    if(*d1>*d2) return 1;
    return 0;
}

int main(void){
    double t[]={2,3,5,7,11,13,17,19,23,27,31,37};
    size_t n=sizeof(t)/sizeof(double);
    double mit=19,*p;
    p=(double*)bsearch(&mit,t,n,sizeof(double),
        hasonlit);
}
```



```

if (p==NULL)printf("Nincs\n");
else printf("Megvan: t[%u]=%g\n",p-t,*p);
return 0;
}

```

Gondolkozzunk együtt!

Oldja meg egyedül!

30. Rendezés

Rendezett adatsorban keresni lényegesen gyorsabb, mint rendezetlenben, ezt láttuk az előző fejezetben. A rendezésnél fontos szempont, hogy az algoritmus a lehetőségekhez képest gyors legyen, és minél kevesebb ideiglenes memóriát foglaljon, azaz az adatokat a saját helyükön rendezze.

Ebben a fejezetben először tömbök rendezésére nézünk meg néhány algoritmust, végül bemutatjuk a gyorsrendező algoritmus két irányban láncolt listán működő változatát is.

A rendezőalgoritmusok egyikére sem lehet általánosságban kijelenteni, hogy mindig ez vagy az a leggyorsabb. Például a winchesteren lévő adatok rendezésére egész más algoritmusok hatékonyak, mint a memóriában lévő adatok rendezésére, vagyis az algoritmus hatékonysága függ a hardver környezettől is.

Egy tömb elemeit sorba rendezni nem ördögötől való dolog, bárki, aki az eddigi részeket nagyjából megértette és tudja használni, képes írni ilyet.

Írjon föl cédulákra 5-10 különböző egész számot, keverje össze a papírokat, és aztán rakja növekvő sorba! A sorba rakásnál ügyeljen arra, hogy a számítógép nem látja egyszerre az összes értéket, csak 1-2 kiválasztottat, tehát ön se lássa egyszerre az összeset, csak azokat, amelyekre az ujjával mutat! Cserélgesse addig a cetliket, amíg azok helyes sorrendbe nem kerülnek!

Az algoritmusokat *double* elemű tömbök rendezésére alkalmas C függvényekkel valósítjuk meg. Az algoritmust egy konkrét tömb rendezésével is bemutatjuk, érdemes először ezt megnézni, és csak ezután a programkódot. Mivel a tömb rendezése nagy helyet foglal az oldalon, az adott rendezéshez kapcsolódó alfejezetben belül különböző helyeken jelenik meg.

30.1 Buborék algoritmus

Egyszerű algoritmus, de nem nagyon szeretjük, mert nem túl hatékony.

Működés:

1. Veszünk két szomszédos tömbelemet, és összehasonlítjuk. Ha a sorrendjük nem megfelelő (azaz növekvő sorba való rendezésnél nem a kisebb van előbb), akkor megcseréljük az elemeket.
2. Egy elemmel tovább lépünk, és így végezzük el az 1. lépésben mondottakat (tehát az előző lépésben összehasonlított két elem közül a másodikat hasonlítjuk most a következő tömbelemmel, lásd az ábrát)
3. Ezt ismétljük, amíg a tömb végére nem érünk. Ekkor az biztos, hogy a legnagyobb tömbelem eljutott a tömb végére. (Más elemek is közelebb kerülhetnek a végső helyükhöz.)
4. Ismétljük az eljárást a tömb elejétől. Most elég az utolsó előtti helyig menni, és így a második legnagyobb tömbelem az utolsó előtti helyig „bugyborékol”.
5. $n-1$ -szer megismételve a végigbugyborékolást a tömb rendezetté válik.

```

//*****
void buborek(double t[],int db){
//*****
int i,j;

for(i=0;i<db-1;i++){
for(j=0;j<db-1-i;j++){
if(t[j]>t[j+1]){
double temp=t[j]; // csere
t[j]=t[j+1];
t[j+1]=temp;
}
}
}
}

```

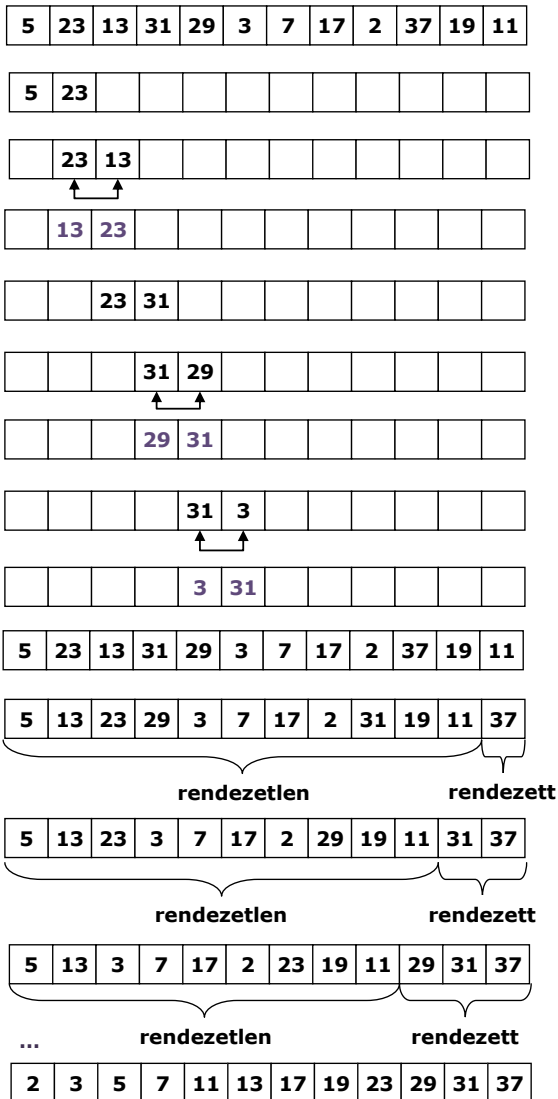
}
Az algoritmus sebessége némileg javítható, ha figyeljük, hogy volt-e csere egy végigbugyborékoltatás alatt. Ha nem, akkor a tömb rendezett, tehát szükségtelen folytatni a rendezést:

```

//*****
void buborek_jav(double t[],int db){
//*****
int i,j,csere=1;

for(i=0; i<db-1 && csere!=0; i++){
csere=0;
for(j=0; j<db-1-i; j++){
if(t[j]>t[j+1]){
double temp=t[j]; // csere
t[j]=t[j+1];
t[j+1]=temp;
csere=1;
}
}
}
}

```



Buborék rendezés:

Összehasonlítások száma:

→ worst case: $O(n^2)$

→ best case: $n-1 = O(n)$

Cserék száma:

→ worst case: $O(n^2)$

→ best case: 0

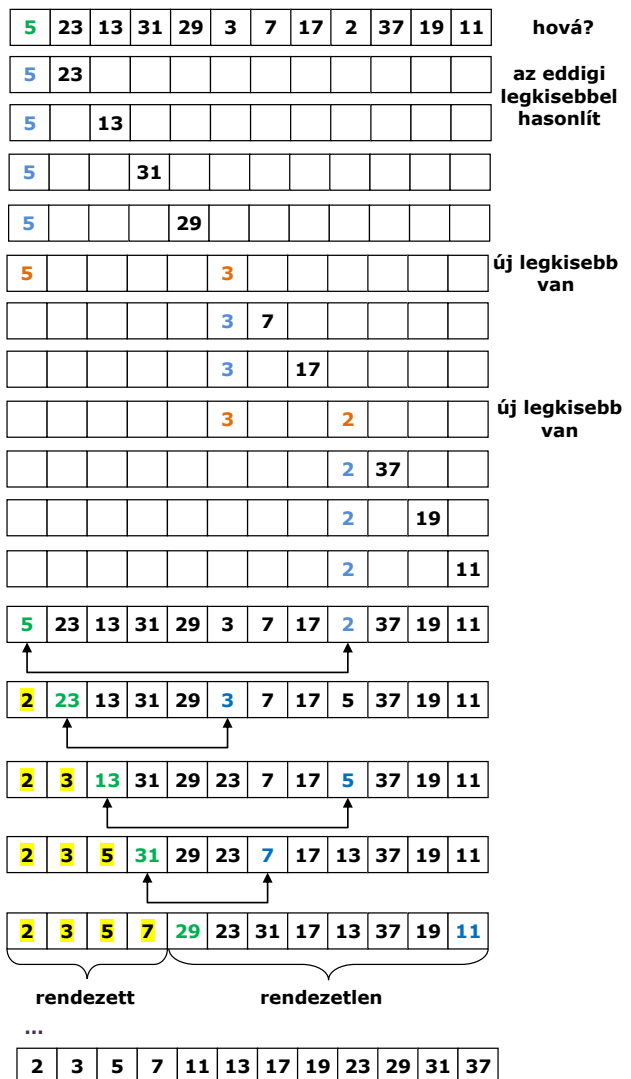
Javított buborék rendezés:

Ugyanaz, mint a buborék, de a várható lépésszám kisebb

30.2 Közvetlen kiválasztásos rendezés

Működés:

1. Jegyezzük meg, hogy a tömb első eleme helyére akarjuk betenni a tömb legkisebb elemét!
2. Keressük meg a legkisebb elemet! Először feltételezzük, hogy az első elem a legkisebb, és ezzel hasonlítjuk össze sorra a tömb elemeit.
3. Ha találunk kisebbet, akkor ettől kezdve ezzel hasonlítunk.
4. Ismétljük a 3. lépést a tömb végéig! Így megtaláltuk a legkisebbet.
5. Cseréljük ki a megjegyzett elemet a legkisebbre, ha nem a megjegyzett volt a legkisebb!
6. Jegyezzük meg, hogy az előzőleg megjegyzett utáni tömbelem helyére akarjuk becserélni a maradék tömbből a legkisebbet, és ismétljük a lépéseket a 2.-tól, míg a megjegyzett tömbelem az utolsó nem lesz!
7. A tömb rendezett.



Közvetlen kiválasztásos rendezés:

Összehasonlítások száma:

→ worst case: $O(n^2)$

→ best case: $O(n^2)$ // rosszabb, mint buborék

Cserék száma:

→ worst case: $n-1 = O(n)$ // jobb, mint buborék

→ best case: 0

```

//*****
void kozvetlen(double t[],int db){
//*****
    int i,j,minindex;

    for(i=0;i<db-1;i++){ // i: hova
        minindex=i;
        for(j=i+1; j<db; j++)
            if( t[j]<t[minindex] ) minindex=j;
        if(minindex!=i){
            double temp=t[minindex]; // csere
            t[minindex]=t[i];
            t[i]=temp;
        }
    }
}

```

```

        t[j] = t[j-1];
        t[j-1] = temp;
    }
}

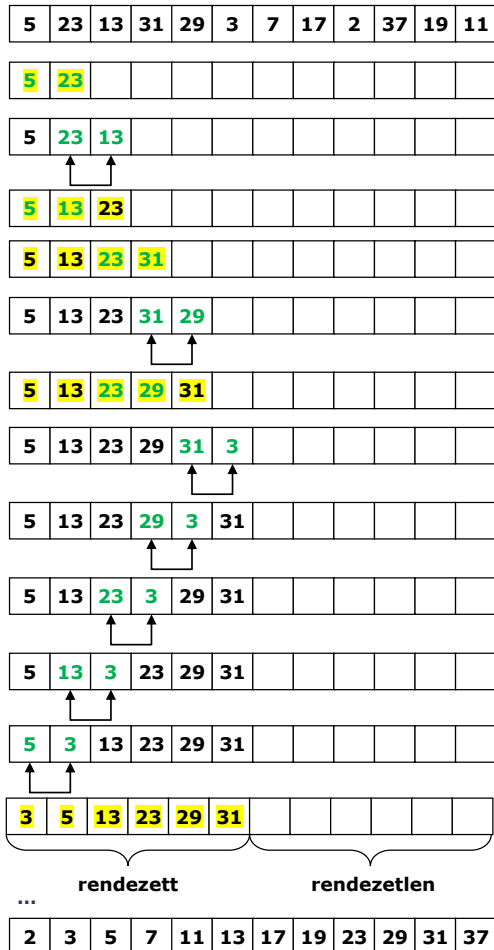
```

Összehasonlítások száma:
 → worst case: $O(n^2)$
 → best case: $n-1 = O(n)$
 Cserék száma:
 → worst case: $O(n^2)$
 → best case: $2*(n-1) = O(n)$

30.3 Süllyesztő rendezés

Működés:

1. A tömb első két elemét megfelelő sorrendbe tesszük.
2. Most a tömb eleje rendezetten tartalmaz elemeket.
3. A következő tömbelemet addig cserélgetjük lefelé, amíg a nála eggyel kisebb indexű elem nála nagyobb. Ekkor ismét rendezett lesz a tömb eleje, csak eggyel több elem van ezen a részen, mint előbb.
4. A 3. lépést ismételjük egész addig, amíg van olyan tömbelem, amit nem süllyesztettünk a helyére.
5. A tömb rendezett.



```

//*****
void sullyesztto(double t[],int db){
//*****
    int i, j;

    for(i=1; i < db; i++){
        for(j=i; j > 0 && t[j-1] > t[j]; j--){
            double temp = t[j]; // csere

```

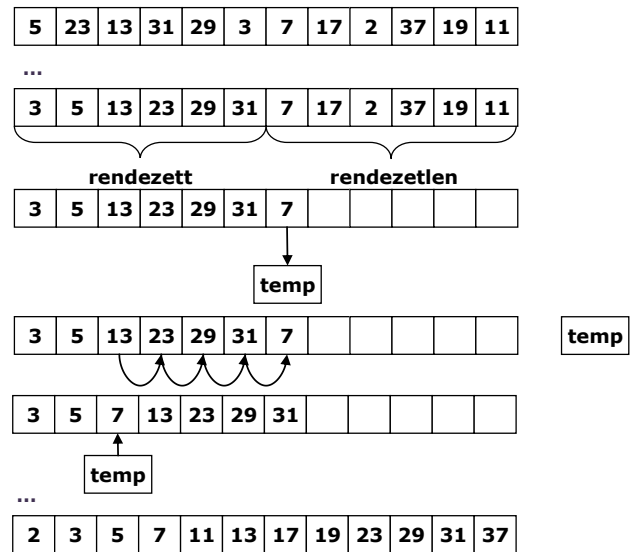
30.4 Közvetlen beszűrő rendezés

A süllyesztő rendezés javított változata.

Működés:

1. Bináris kereséssel megkeressük a tömb már rendezett részén a nem rendezett rész első elemének helyét. Az elemet ideiglenes változóba kivesszük.
2. A beszűrés helyétől az elemeket eggyel feljebb mozdítjuk. (Figyeljük meg, hogy a $t[j]=t[j-1]$ másolásnál j értéke csökken a ciklusban. Ha nőne, ugyanazzal az elemmel íránk felül az összes elemet a rendezett rész végéig.)
3. Az ideiglenes változóból betesszük az elemet a helyére.
4. Addig ismételjük 1-3-ig, míg a tömb rendezett nem lesz.

Így az összehasonlítások száma jóval kevesebb lehet, a cserék száma viszont változatlan.



```

//*****
void beszuro(double t[],int db){
//*****
    int i,j;

    for (i=1; i<db; i++){
        int min=0, max=i-1, med;
        double temp=t[i];

        while(min<=max){ // bin keres
            med=(min+max)/2;
            if(temp<t[med]) max=med-1;
            else min=med+1;
        }
        for(j=i; j>min; j--) t[j]=t[j-1]; // eltol
        t[min]=temp;
    }
}

```

Közvetlen beszűrő rendezés:

Összehasonlítások száma:

→ worst case: $O(n \cdot \log n)$ // jobb, mint az előzőek

→ best case: $O(n \cdot \log n)$

Cserék száma:

→ worst case: $O(n^2)$ // a közvetlen kiválasztás jobb

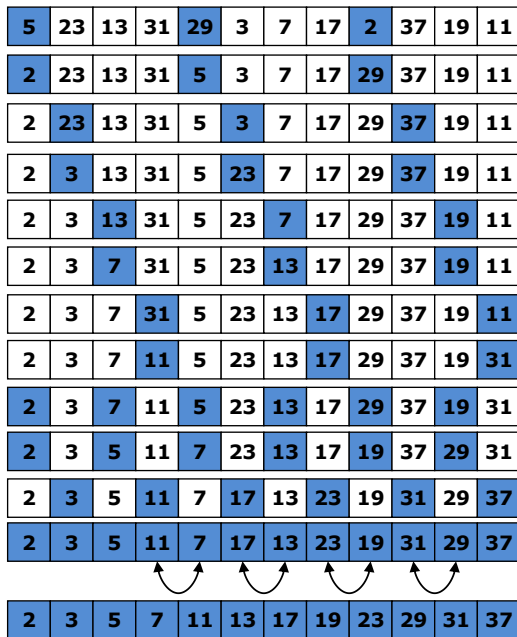
→ best case: $2 \cdot (n-1) = O(n)$

30.5 Shell rendezés

Donald Shell dolgozta ki. Cél: az elemek mielőbb végleges helyük közelébe jussanak.

Működés:

1. Bontjuk a tömböt sok kevés elemből álló részre (Pl. 3 elemű részre). A részre bontást nem szomszédos elemek alkotják, hanem egymástól nagy, egyenlő távolságra lévő elemek.
2. Rendezzük az elemeket az egyes részeken belül! (Pl. $n/3$ db tömböt rendezünk így.) A rendezéshez gyakorlatilag bármilyen megfelelő algoritmus használható, a mintakódban süllyesztő rendezés szerepel.
3. A részre bontás rendezése kevés mozgással jár, de az elemek nagyot ugrottak a majdani végső helyük felé.
4. Most az előzőnél sűrűbb részre bontást jelöljük ki (a mintakódban háromszorosára sűrítjük a részre bontást, a rajzon csak kétszeresére, mert így jobban látható, mit csinálunk.)
5. Rendezzük az új részre bontást a 2. pontban írtak szerint.
6. Ismétljük a 4-5 lépéseket addig, amíg egyetlen részre bontásunk nem lesz, azaz maga a teljes tömb. Ha ezt rendezzük, sokkal kevesebb mozgásra lesz szükség, mintha az eredeti tömböt rendeztük volna a süllyesztő algoritmusmal.



Shell rendezés:

Összehasonlítások száma:

→ worst case: $O(n^A)$, $A=1,2,\dots,1,5$

→ best case: $O(n^A)$

Cserék száma:

→ worst case: $O(n^A)$, $A=1,2,\dots,1,5$

→ best case: 0

```

//*****
void Shell(double t[],int db){
//*****

```

```

int i,j,dist;
for(dist=(db+1)/3; dist>0; dist=(dist+1)/3){
for(i=dist; i<db; i+=dist)// süllyesztő rend.
if(t[i]<t[i-dist]){
double temp=t[i];
for(j=i-dist;j>=0&& t[j]>temp;j-=dist)
t[j+dist]=t[j];
t[j+dist]=temp;
}
}
}

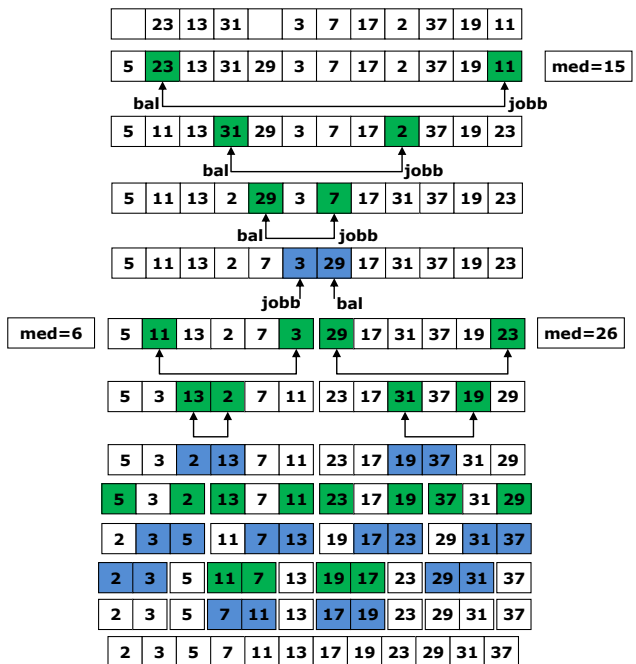
```

30.6 Gyorsrendezés

A Shell algoritmushoz hasonlóan ez is gyorsan a végleges helyre közelébe viszi az elemeket.

Működés:

1. Tippeljünk meg, mennyi lehet a tömb középső eleme (medián)! (Pontosan nem tudjuk, mert ahhoz rendezni kéne a tömböt.) Az adattípustól függően ez lehet pl. a két szélső adat átlaga, vagy valamelyik szélső adat. (Pl. sztringtömb esetén nehéz lenne átlagot számolni...)
2. A mediánál kisebb elemeket „dobáljuk” a tömb elejére, a mediánál nagyobbakat pedig a tömb végére! (Azaz a két féltömb nem lesz rendezett, csak az lesz biztos, hogy az alsó felében minden elem kisebb lesz, mint bármelyik elem a felső felében.)
3. A „szétdobálás” a következőképp történik:
 - a. Induljunk balról, és keressük meg az első olyan elemet, amelyik nagyobb a mediánál, tehát a tömb rossz oldalán van.
 - b. Induljunk jobbról, és keressük meg az első olyan elemet, amelyik kisebb a mediánál, tehát a tömb rossz oldalán van.
 - c. Ha a bal elem balra van a jobb elemtől, cseréljük meg őket, és ismétljük az a-b lépéseket innen továbbindulva.
 - d. Ha a bal elem jobbra van a jobb elemtől, akkor a tömb elemei szét vannak választva.
4. Ismétljük az 1-4 lépéseket a két részre bontásra! (A 4. pont miatt **rekurzív** a folyamat.)
5. Addig megy a folyamat, amíg egyelemű tömbökig nem jutunk. Ekkor befejezzük, mert kész a rendezés.



```

//*****
void gyors(double t[],int db){
//*****

```

```

int bal=0, jobb=db-1;
double med=(t[bal]+t[jobb])/2; // nem ideális

if(db<2) return; // rekurzió stop
do{
    while(t[bal]<med) bal++;
    while(t[jobb]>med) jobb--;
    if(jobb>=bal){
        double temp=t[bal]; // csere
        t[bal]=t[jobb];
        t[jobb]=temp;
        bal++; jobb--;
    }
}while(jobb>=bal);
if(jobb>0) gyros(t, jobb+1);
if(bal<db-1) gyros(t+bal, db-bal);
}

```

Összehasonlítások száma:

- worst case: $O(n^2)$
- best case: $O(n \cdot \log_2 n)$
- átlag: $O(n \cdot \log_2 n)$

Cserék száma:

- worst case: $O(n^2)$
- best case: 0
- átlag: $O(n \cdot \log_2 n)$

30.7 Szabványos könyvtári gyorsrendező függvény: a qsort

Az stdlib.h-ban található a

```

void qsort(
    void *base,
    size_t nelem,
    size_t width,
    int (*fcmp)(const void *elem1, const void *elem2)
);

```

prototípusú tömbrendező függvény.

base: a tömb címe

nelem: elemszám

width: egy elem mérete

fcmp: két tömbelemet összehasonlító függvény címe

Pl.:

```

double t[7]={863, -37, 54, 9520, -3.14, 25, 9999.99};
qsort(t, 7, sizeof(double), hasonlit);

```

```

int hasonlit(const void *a, const void *b){
    double *ia=(double*)a;
    double *ib=(double*)b;
    if(*ia<*ib) return -1; // <0
    if(*ia==*ib) return 0; // ==0
    return 1; // >0
}

```

A hasonlító függvény visszatérési értéke <0 egész szám, ha az első pointer által mutatott tömbelemnek a tömbben hártébb kell állnia rendezés után, >0, ha előbb kell állnia (azaz csere szükséges), és 0, ha a kettő sorrendje mindegy (pl. ha egyenlőek).

Ezen a módon tetszőleges típusú elemeket tartalmazó tömbök rendezhetők. Pl. sztringekre mutató pointerok tömbje esetén az *strcmp* függvény visszatérési értéke pont az, ami kell.

Struktúrátömb esetén pl.:

```

typedef struct _hallgato{
    char nev[50];
    char neptun[7];
    double atlag;
}hallgato;

int hasonlit1(const void *a, const void *b){
    hallgato *x=(hallgato*)a;
    hallgato *y=(hallgato*)b;
    return strcmp(x->nev, y->nev);
}

```

```

int hasonlit2(const void *a, const void *b){
    hallgato *x=(hallgato*)a;
    hallgato *y=(hallgato*)b;
    if(x->atlag < y->atlag) return -1;
    if(x->atlag == y->atlag) return 0;
    return 1;
}

qsort(t, 100, sizeof(hallgato), hasonlit1);
qsort(t, 100, sizeof(hallgato), hasonlit2);

```

Az első rendezés név szerint teszi sorba a tömböt, a második pedig átlag szerint.

30.8 Láncolt lista rendezése gyorsrendező algoritmussal

Az összes olyan rendező algoritmust át lehet írni duplán láncolt listára, ahol nem ugrálunk a tömbben, hanem szomszédos elemeket veszünk. Egyirányban láncolt lista esetén követelmény még, hogy csak előre kelljen haladni az elemek között.

Egyirányú lista rendezésére alkalmas az eddigi algoritmusok közül: buborék, közvetlen beszúrás

Kétirányú lista rendezésére alkalmas ezen felül még: a süllyesztő, gyorsrendezés

A közvetlen beszűrő nem volt jó a bináris keresés miatt. A Shell csak nagyon nehézkesen lenne használható, inkább ne próbálkozzunk vele.

Gyorsrendezés listán:

```

//*****
typedef struct _szam{
//*****
    int n;
    struct _szam *prev, *next;
} szam;

//*****
void csere(szam * p1, szam * p2);
//*****

//*****
void lista_qsort(szam * elso, szam * utolso){
//*****
    szam *bal=elso, *jobb=utolso;
    int med=(elso->n+utolso->n)/2;

    if(elso==utolso) return; // rekurzió stop
    do{
        while(bal->n<med) bal=bal->next;
        while(jobb->n>med) jobb=jobb->prev;
        if(jobb->n>=bal->n){
            szam * temp;
            csere(bal, jobb);
            temp=bal; bal=jobb; jobb=temp;
            if(bal!=jobb) bal=bal->next;
            if(bal!=jobb) jobb=jobb->prev;
        }
    }while(bal!=jobb && jobb->n>=bal->n);
    lista_qsort(elso, jobb);
    lista_qsort(bal, utolso);
}

```

A *csere* függvény definíciója a 24.3 fejezetben található.

Gondolkozzunk együtt!

Oldja meg egyedül!

Házi feladatnak (nagyon sok apróságra oda kell figyelni a megoldás során): Írjon függvényt, amely paraméterként kap egy szóközzel elválasztott szavakat tartalmazó sztringet, és

ABC sorba rendezi a szavakat a sztringen belül! A szavak hossza 1 és 25 karakter között bármennyi lehet. Maximum három akkora segéd tömböt használhat, melyben egy-egy szó elfér (nem biztos, hogy szüksége lesz mindre), további segéd tömböt nem használhat!

31. Preprocesszor

Mielőtt egy C program a fordítóhoz kerülne, átmege rajta az előfeldolgozó. Ez egy egyszerű szövegszűrő és –behelyettesítő alkalmazás. Az a célja, hogy a fordítónak ne kelljen a C program formai dolgaival foglalkoznia, csak a fordítással törődhessen. Eltávolítja a megjegyzéseket a kódból, a whitespace karaktereket optimalizálja, és végrehajtja a preprocesszor utasításokat (sor eleji # jelzi ezeket, és végüket nem zárja ;, a sor végéig tartanak). A legfontosabb preprocesszor utasítások a következők:

#define ► szimbólum, konstans és makró definícióra

szimbólum:

```
#define SZIMBOLUM
```

- definiálja a SZIMBOLUM azonosítót.
- Önmagában csak arra jó, hogy később meg lehessen állapítani, hogy van ilyen szimbólum.
- A szimbólumra ugyanazok a szabályok érvényesek, mint a többi C azonosítóra: az angol ABC kis- és nagybetűiből, számokból és aláhúzás jelekből állhatnak, számmal nem kezdődhetnek. Az a szokás, hogy kisbetűt nem használnak, így könnyen megkülönböztethetők a többi azonosítótól.

konstans:

```
#include <stdio.h>

#define KONSTANS 10
#define KONSTANS2 "Szöveg"
#define EGYESIT for(i=0;i<KONSTANS;i++) tomb[i]=1.0;
#define KIIR
for(i=0;i<KONSTANS;i++) printf("%g\n",tomb[i]);

int main(void){
    double tomb[KONSTANS];
    int i;
    for(i=0;i<KONSTANS;i++) tomb[i]=0.0;
    KIIR
    EGYESIT
    KIIR
    puts(KONSTANS2);
    puts("KONSTANS2"); // nem helyettesít
    return 0;
}
```

- Ahol az előfeldolgozó megtalálja a konstanst, oda behelyettesíti a megadott, sor végéig tartó szöveget, ami akár utasítás is lehet. A kódban szereplő konstans sztringbe nem helyettesít.)
- Ha ;-t teszünk a sor végére, azt is behelyettesíti a kódba.
- Ha hibásan adjuk meg a definiált konstanst (vagy makrót), akkor a fordító nem a konstans definíciójánál jelzi a hibát, hanem a behelyettesítés helyén. (Ha pl. `#define KONSTANS 10;` szerepelne, a VC fordító hét hibaüzenetet ad, pl.: `syntax error : missing ']' before ';'.` Nem könnyű rájönni, mi is a baj.)

makró:

```
#include <stdio.h>

#define INT_KIIR(a) printf("%d\n",a)
#define MIN(a,b) ((a)<(b)?(a):(b))

int main(void){
    int a=0,b=10;
    INT_KIIR(6);
    INT_KIIR(MIN(++a,++b));
    // baj van: printf("%d\n",((++a)<(++b)?(++a):(++b)))
    return 0;
}
```

- Függvényre hasonlít.

- Az előfeldolgozó szövegszerű behelyettesítést végez, a fordító a behelyettesített kódot kapja, ami gondokat okozhat. Pl. a kódra ránézve azt hisszük, hogy *a* és *b* is eggyel nő, és a behelyettesítés után látjuk, hogy *a* kettővel nőtt.

Gondolkozzunk együtt!

#undef ►

```
#undef MIN
```

Szimbólum, konstans, makró törlése. Akkor hasznos, ha pl. lesz utána #ifdef vagy újra akarjuk definiálni.

Oldja meg egyedül!

Feltételes fordítás ►

#if kifejezés ... #else // opc ... #endif	#if kifejezés ... #elif kifejezés ... #endif
#ifdef azonosító ... #else // opc ... #endif	#ifndef azonosító ... #else // opc ... #endif

A *kifejezés* fordítási időben kiértékelhető legyen (azaz konstans). *elif* = *else if*

#error ► Fordítási hibaüzenetet adhatunk.

```
#ifdef MIN
#error MIN redefinition.
#endif
```

Előre definiált preprocesszor konstansok ►

__DATE__:	Az adott forrásfájl fordításának dátuma <i>Mmm dd yyyy</i> formában.
__TIME__:	Az adott forrásfájl fordításának időpontja <i>hh:mm:ss</i> formában.
__TIMESTAMP__:	Az adott forrásfájl utolsó módosításának dátuma és időpontja <i>Ddd Mmm Date hh:mm:ss yyyy</i> formában.
__FILE__:	Az adott forrásfájl neve.
__LINE__:	Az adott forrásfájl aktuális sorának sorszáma egész konstansként.

#include ► Az utána megadott nevű fájlt beszerkeszti az adott helyre.

A `<>` között megadott fájlt a fordító beállításainál beállított fejléc mappákban keresi, az `""` között megadott fájlt pedig a fordítandó program mappájába.

A fájlnev útvonalat is tartalmazhat, pl.: `<sys/stat.h>`, `../programok/headerk/lacigrafika.h`. Nem csak fejlécfájl szerkeszthető be, hanem bármi. `#include`-ot nem csak a fájl elejére tehetünk, hanem bármelyik sorába, a fordító oda fogja beszerkeszteni.

#pragma ► Fordítószerkesztési fordítási beállítások megadását teszi lehetővé, ezek között tehát nincs olyan, ami minden C fordítóval megy.

Visual C-ben talán a legfontosabb a `#pragma pack(1)` beállítás: ezzel érhetjük el, hogy a struktúrák/unionok adatai között ne legyen „hézag” a memóriában. Alapesetben a fordító 8 bajtonként helyezi el az adatokat a gyorsabb műveletvégzés érdekében. 1-re állítani akkor lehet értelme, ha ilyen formátumú bináris fájl kell írni/olvasni, pl. ha *bmp* fájlt kezelünk.

A további `#pragma` lehetőségekről a fordító HELP-jéből tájékozódhatunk.

32. Bitmezők, union

Két, korábban nem részletezett, kevésbé fontos adattípus.

32.1 Struktúra bitmezői

Ha 1 bájt nál kisebb is elég egy változónk tárolásához, vagy mondjuk 12 bit is elég, akkor használhatjuk a struktúra bitmezőit. A módszer akkor is hasznos, ha hardvert akarunk programozni. A bitmezőkkel elérhető eredmények a bitszintű operátorokkal is megoldhatók, de a bitmezők használata kényelmesebb.

```
#include <stdio.h>

struct bitmezos{
    int a : 4;
    int b : 12;
    double d;
};

int main(void){
    struct bitmezos b;
    b.a=10;
    b.b=10;
    b.d=b.a*b.b;
    printf("%d, %d, %g\n",b.a,b.b,b.d);
    return 0;
}
```

- A bitmezőnél használt alaptípus csak *int*, *signed int* vagy *unsigned int* lehet.
- Az *a* adattag 4 bites előjeles egész számot tárol, azaz -8..+7 közötti értéket. 10-et adva neki, túlsordulás következik be, hibás érték (-6) lesz benne. A *b* már 12 bites, így gond nélkül tárolja a 10-et.
- Ha *a* típusa *unsigned* lenne, akkor már a 10-et is jól tárolná, hiszen 4 biten 0..15-ig férnek a számok.
- Mivel ezt a tárolást a számítógép bitenkénti műveletek felhasználásával valósítja meg, lassabb lesz a működés, mint pl. normál *int* vagy *char* típus használata esetén lenne.
- A nagy helytakarékosságban ne feledjük, hogy a fordítóprogram a gyorsabb működés érdekében gyakran szóhatárra (2, 4, 8, 16 bájtontként, beállításfüggő) illeszti a struktúrák tagváltozóit. A fenti kódot Visual C-ben lefordítva, ha megnézzük, mennyi *sizeof(b)*, 16-ot kapunk, mert *a* és *b* ugyanabban az *int* típusú változóban van. Ezt a változót egy 4 bájtos felhasználatlan üres hely követi a memóriában, majd következik a *d* változó. vagyis ha *a* és *b* sima egész lett volna, ugyanennyi helyet foglalna a struktúra. Ha *d*-t elhagyjuk, és csak *a* és *b* marad, akkor 4 bájt lesz a struktúra mérete. Visual C-ben a szóhatárt 1 bájtra állíthatjuk a *#pragma pack(1)* preproceszor utasítással, bővebben lásd a Helpben.

32.2 Union

Az union adattípus formailag megegyezik a struktúrával, máshogy működik azonban:

```
#include <stdio.h>

union uni{
    unsigned char t[100];
    int i;
    double d;
};

void kiir(union uni x){
    int i;
    printf("char: ");
    for(i=0;i<8;i++)printf("%hhu", x.t[i]);
    printf("\nint=%d\ndouble=%.20g\n", x.i, x.d);
}
```

```
}

int main(void){
    union uni x;
    int i;
    for(i=0;i<100;i++)x.t[i]=0;
    kiir(x);
    x.i=55;    kiir(x);
    x.d=88.0;  kiir(x);
    x.i=55;    kiir(x);
    x.i=500;   kiir(x);
    x.d=88.1;  kiir(x);
    return 0;
}
```

A program futásának eredménye Intel architektúrájú processzoron, 32 bites rendszerben:

```
char: 0,0,0,0,0,0,0,0,
int=0
double=0

char: 55,0,0,0,0,0,0,0,
int=55
double=2.717361052126856e-322

char: 0,0,0,0,0,0,86,64,
int=0
double=88

char: 55,0,0,0,0,0,86,64,
int=55
double=88.0000000000000782

char: 244,1,0,0,0,0,86,64,
int=500
double=88.0000000000007105

char: 102,102,102,102,102,6,86,64,
int=1717986918
double=88.099999999999994
```

Magyarázat:

- Az alább leírtak csak 32 bites Windowsban, VisualC fordítóval vannak így. Más architektúrán más lehet az eredmény, azaz a kód nem platformfüggetlen!
- A *union* típusú változóban lévő tagváltozók fizikailag ugyanazon a memóriahelyen található, ami azt jelenti, hogy ha megváltoztatjuk az egyiket, az összes többi is megváltozik.
- Először kinullázzuk a karaktertömböt, ez egyúttal a másik két változó kinullázását is jelenti, hiszen azok kisebbek 100 bájtnál.
- Ezután az egész számba 55-öt írunk. Kiírásnál az *int* változóban 55.
 - A karaktertömb első bájtja szintén 55. Elsőre talán nem tűnik fel, de ez nem nyilvánvaló. Azt várnánk, hogy a karaktertömb első 8 eleme 0,0,0,55,0,0,0,0 legyen. Az Intel architektúrájú processzorok azonban az adatokat fordított bájtsorrendben tárolják.
- A *double* érték nem 55! Jól jegyezzük meg, hogy ezzel a módszerrel nem lehet típuskonverziót végrehajtani. A kapott érték az a *double* szám, amelynek megegyezik a bitmintája a karaktertömbnél látható bitmintával.
- A *double* értéket 88-ra állítjuk. A fordított bájtsorrend miatt az *i* értéke 0.
- Az *i*-be ismét 55-öt írunk. Mivel az *i* csak feleakkora helyet foglal a memóriából, mint a *d* (32 bites a 64-gyel szemben), csak a *d* felét változtatja meg, *d* másik fele

nem változik, és mivel abban a felében volt az érdemi rész, azt látjuk, hogy alig rontottuk el a *d*-ben lévő 88-at.

- 500-at írva látjuk, hogy a karaktertömb *i* által lefoglalt részén már 2 hely is különbözik 0-tól. 500 könnyedén kiolvasható: $1 \times 256 + 244$.
- A 88.1 nem ábrázolható pontosan kettes számrendszerben, így a szám vége nem csupa 0.

A *union* adattípust ritkán használjuk.

Gondolkozzunk együtt!

Oldja meg egyedül!

33. Változó paraméterlistájú függvények

A C nyelvben talákoztunk olyan függvényekkel, melyeknek bármennyi paramétere lehetett (*printf* és *scanf* családok). Most megnézzük, hogyan lehet ilyet létrehozni.

Változó argumentumszámú függvények létrehozásához az *stdarg.h* fejlécállomány beszerkesztése szükséges. Ebben vannak definiálva a makrók, valamint a *va_list* pointertípus. A változó argumentumszámú függvényeknek legalább egy konkrét típusú paraméterrel kell rendelkeznie, ezt követi az opcionális paraméterek listája, melyet ... jelöl. Ezt már nem követhetik további konkrét paraméterek, azaz a ... mindig a lista végén áll.

Az argumentumlista kezeléséhez három makró használandó:

- *void va_start(va_list arg_ptr, prev_param)*: a lista feldolgozásakor ezzel kezdünk. A makró második paraméterként kapja az első opcionális argumentumot megelőző változót, és beállítja az első paraméterként kapott pointert az opcionális argumentum kezeléséhez megfelelően. (Mivel makróról van szó, nem a pointer címét kell átadni, hanem magát a pointert.) Ha belenézünk az *stdarg.h*-ba, ehhez hasonló megvalósítást találunk: $arg_ptr = \&prev_param + 1$
- *type va_arg(va_list arg_ptr, type)*: a makró visszaadja a következő opcionális paraméter értékét. Ha belenézünk az *stdarg.h*-ba, ehhez hasonló megvalósítást találunk: $*((type*)(arg_ptr = (type*)arg_ptr + 1) - 1)$
- *void va_end(va_list arg_ptr)*: ha befejeztük a lista feldolgozását. Ha belenézünk az *stdarg.h*-ba, ehhez hasonló megvalósítást találunk: $arg_ptr = NULL$

A következő példaprogram kiírja a paraméterként átvett értékeket a formátumsztringnek megfelelően. A formátumsztring három karaktert tartalmazhat: 'g'-t, 'd'-t és 's'-t. A 'g' double típust jelöl, a 'd' int-et, az 's' pedig sztringet. Az argumentumokat külön sorba írja, és kiírja az argumentum sorszámát is.

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

void unikiir(char t[],...){
    va_list p;
    int i;

    va_start(p,t);
    for(i=0; t[i]!='\0'; i++){
        switch(t[i]){
            case 'd':
                printf("%d. %d\n",i+1,va_arg(p,int));
                break;
            case 'g':
                printf("%d. %g\n",i+1,va_arg(p,double));
                break;
            case 's':
                printf("%d. %s\n",i+1,va_arg(p,char*));
                break;
        }
    }
    va_end(p);
}

int main(){
    unikiir("gdsd",3.14,-5,"Hello",3);
    return 0;
}
```

Hibakezelő függvény ► A következő függvény a *printf*-hez hasonlóan használható, de kiírja, hogy „Error:” a megjelenítendő szöveg elé, továbbá a kiírás után kilép a programból. A *vfprintf*-hez hasonlóan a többi *printf* függvénynek is van ilyen „továbbdobott” változata.

```

void error(const char *s,...){
    va_list p;

    va_start(p,s);

    fprintf(stderr,"\n\nError: ");
    vfprintf(stderr,s,p);
    fprintf(stderr,"\n\nPress ENTER to continue...");

    va_end(p);
    scanf("%*s");
    exit(1);
}

if(p==NULL)
    error("Cannot open file (%s) in line %d of %s"
        ,filename,__LINE__,__FILE__);

```

Gondolkozzunk együtt!

Oldja meg egyedül!

34. Hasznos függvények a C-ben

Álvéletlen számok ► Az `stdlib.h` fejléc beszerkesztésével érhetjük el a `int rand(void)`; prototípusú függvényt, mely a $0..RAND_MAX$ közötti álvéletlen számot ad vissza. `RAND_MAX` értéke az ISO C szabvány szerint legalább 32767.

Amennyiben $0..n$ közötti egész számra van szükség véletlen számként, a `rand()%n` kifejezést alkalmazhatjuk.

Ha lebegőpontos véletlen szám szükséges: `double d=(double)rand()/RAND_MAX` módon kaphatunk 0..1 közé eső értéket.

Ha a `rand()` függvényt meghívva minden programindításkor ugyanazt a számsort kapjuk. Ezt elkerülhetjük a véletlen szám generátor kezdőértékének átállításával, melyet a

```
void srand( unsigned seed );
```

prototípusú függvény meghívásával tehetünk meg. Ahhoz, hogy a kezdőérték mindig más lehessen, a függvénynek a rendszeri időt adjuk kezdőértékként:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(){
    int i;

    srand( (unsigned)time(NULL) );
    for(i=0; i<10; i++)printf("%d\n",rand());
}

```

Fájlok törlése, átnevezése ► Az `stdio.h`-ban található függvényekkel lehetséges:

- `int remove(const char *path)`: a `path` sztring a fájl nevét tartalmazza, útvonalat is meg lehet adni. A visszatérési érték 0, ha sikerült.
- `int rename(const char *oldname, const char *newname)`: az `oldname` nevű fájlt vagy mappát `newname` névre nevezi át. A visszatérési érték 0, ha sikerült. Fájl másik mappába is helyezhető, ha a `newname`-ben megadott útvonal más, mappa viszont nem helyezhető át.

Időkezelés a C-ben ► A `time_t time(time_t *timer)`; függvény az 1970. január 1. 00:00:00 óta eltelt időt adja vissza másodpercben, a `time_t` típus tehát egy `typedef`-fel létrehozott egész típus. Értéke nem definiált 2038. január 18. 19:14:07 után. Ugyanazt az értéket adja visszatérési értéként, amit paramétersoron. Paramétersoron NULL is adható.

A `time` által visszaadott időt több függvénnyel is emészthető alakúvá tehetjük.

- `gmtime()`: struktúrára mutató pointert ad vissza, melyben külön van választva év, hónap, nap, perc, másodperc, a hét melyik napja, stb.
- `localtime()`: ugyanolyan struktúra címét adja vissza, mint a `gmtime()`, csak az időzónának megfelelően.
- `asctime()`: sztringgé konvertálja az előző két függvény struktúráját
- `ctime()`: sztringgé konvertálja a `time()` által visszaadott időpontot. A sztring formája pl.: „Wed Jan 02 02:03:55 1980\n0”

A processz indítása óta felhasznált időt kérdezhetjük le a `clock_t clock(void)`; függvénnyel. A `clock_t` egy egész típus. A függvény másodpercnél sokkal pontosabban számol. Programunk sebességét mérhetjük a következő módon:

```

clock_t start, stop;
double eltelt_ido;

start = clock();
... // itt van a mérendő rész

```

```
finish = clock();
eltelt_ido=(double)(stop-start)/CLOCKS_PER_SEC;
printf("%g sec\n",eltelt_ido);
```

A `CLOCKS_PER_SEC` konstans értéke Visual C-ben 1000. Ez nem jelenti azt, hogy a program valóban képes ezred másodperc pontossággal mérni, csak annyit, hogy a `clock()` ezredmásodpercben adja vissza az időt.

A `clock()` függvény a program tényleges futási idejét méri, ami azt jelenti, hogy ha a számítógépen több program generál nagy terhelést a CPU számára, és pl. a programunk a CPU-t 50%-ig terheli, akkor 10 másodperces futási időt a `clock()` 5 másodpercrek fog jelezni. Ha programunk többszálú, a `clock()` függvény másként viselkedik Windows és másként Linux alatt. A Linux magonként számolja a terhelést, azaz ha egy kétmagos gépen futtatjuk a programot, és a program mindkét magot 100%-ra terheli, akkor egy 10 másodperces futási időt a `clock()` 20 másodpercrek fog mutatni, ezzel szemben Windowsban 10 másodpercrek.

Gondolkozzunk együtt!

Oldja meg egyedül!

35. Programozási érdekességek

35.1 Programozás több szálon

A C szabvány nem biztosít lehetőséget a többszálú programok írására, azonban elérhetőek platformfüggetlen függvénykönyvtárak erre a célra. A UNIX rendszerek részét képező POSIX Threads (**pthread**) könyvtárral ismerkedünk meg röviden, melynek 32 és 64 bites Windowsra írt változatai elérhetők pthreads-w32 néven (<http://sourceware.org/pthreads-win32/>). A könyvtár használatához a letöltött kódot le kell fordítani, vagy használható az eleve lefordított változat is.

Windowsban a többszálúnak kívánt programhoz hozzá kell szerkeszteni normál esetben a pthreadVC2.lib könyvtárat (Properties => Linker => Input => Additional dependencies), a pthreadVC2.dll pedig legyen a program mappájában vagy a windows/system32 mappában (esetleg más olyan helyen, ahol a program megtalálja). A programhoz hozzá kell adni a `pthread.h` fejlécállományt. Hogy ez használható legyen, a Properties => C/C++ => General => Additional Include Directories sorban adjuk meg a fejlécet tartalmazó mappát is (ez a header fájl más headereket is be akar szerkeszteni). Ezekon kívül a Properties => C/C++ => Code Generation => Runtime Library-nél Multi-threaded (Multi-threaded debug) könyvtárat állítsunk!

UNIX-ban nem kell semmit tenni, a pthread.h fejléc rendelkezésre áll.

1. Példa (minden példa forrás:

<http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>)

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

void *print_message_function( void *ptr );

main()
{
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int iret1, iret2;

    /* Create independent threads
       each of which will execute function */

    iret1 = pthread_create( &thread1, NULL,
        print_message_function, (void*) message1);
    iret2 = pthread_create( &thread2, NULL,
        print_message_function, (void*) message2);

    /* Wait till threads are complete
       before main continues. Unless we */
    /* wait we run the risk of executing
       an exit which will terminate */
    /* the process and all threads before
       the threads have completed. */

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    exit(0);
}

void *print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

Ahol:

```
int pthread_create(pthread_t * thread, const pthread_attr_t *
attr, void * (*start_routine)(void *), void *arg);
```

- *thread*: a szál azonosítója (olyan, mint a fájlpontor vagy a lefoglalt memória címe: innentől ezzel lehet hivatkozni a szála)
- *attr*: attribútumok, különböző tulajdonságokat állíthatunk be. Ha NULL, default.
- *void * (*start_routine)(void *)*: ezt a függvényt futtatja külön szálon
- *arg*: a függvénynek átadott paraméterek (ha több paramétert akarunk, hozzunk létre struktúrát, és annak a címét adjuk át)

Amikor a fenti programban lefut a *pthread_create* függvény, akkor létrejön egy új programszál, és elindul a megadott függvény, jelen esetben a *print_message_function()*. A *pthread_create* visszatér, *iret1*-be bekerül a visszaadott érték (0, ha nem volt gond), és jön a második *pthread_create* függvényhívás a *message2* paraméterrel, de ezzel párhuzamosan, a háttérben fut az előzőleg elindított *print_message_function()* a *message1*-gyel! A második *pthread_create* elindítja a *print_message_function()*-t a *message2* paraméterrel újabb szálként, így a programunk már 3 szálon fut egyszerre (a két *print_message_function()*, valamint a *main* függvény).

A *pthread_join* függvény addig vár, amíg a paraméterként adott programszál véget nem ér, azaz a *print_message_function()* függvényből vissza nem térünk a kiírás után. Második paramétere za a pointer, amely a függvény által visszaadott pointert tárolja. Mindkét szál befejeződését megvárjuk.

Elképzelhető, hogy a második *print_message_function()* által kiírt „Thread 2” szöveg jelenik meg előbb a képernyőn, hiszen a két szál párhuzamosan fut, közöttük semmiféle erőssorrend nincs rögzítve.

A két szál befejeződése után írjuk ki *iret1* és *iret2* értékét.

A *void pthread_exit(void *retval)*; függvénnyel kiléphetünk a szálból, pl. ha valami hiba történt. (Ha csak egy függvény fut, akkor abból *return*-nel is kiléphetünk, de ha abból más függvényt hívunk, és a hiba ebben a más függvényben keletkezik, akkor innen egyszerűbb a *pthread_exit()* használata.

Szálszinkronizálás ► A Pthreads könyvtár három módszert biztosít a szálak szinkronizálására:

- Mutexek (mutex=MUTual EXclusion lock=kölcsönös kizárás). Blokkolhatjuk más programszálak hozzáférést a védett erőforrásokhoz.
- Join: várakozás más szálak befejeződésére
- Feltétel változók a *pthread_cond_t* típusal megadva

Mutex ► Egyszerű példa mutex nélkül:

```
int counter=0;

void functionC(){
    counter++
}
```

Mutexszel:

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter=0;
void functionC(){
    pthread_mutex_lock( &mutex1 );
    counter++
    pthread_mutex_unlock( &mutex1 );
}
```

A mutex nélküli esetben, ha a *functionC* függvény több példányban fut, akkor megtörténhet, hogy mindkét példány egyszerre olvassa be a *counter* változó értékét, majd mindkettő megnöveli, és mindkettő visszaírja. Ez azt jelenti, hogy *counter* értéke csak eggyel nő, pedig két növelés is történt.

A mutexes változat a következőképp működik:

- Az egyik szál a *pthread_mutex_lock()* függvénnyel *mutex1* változó értékét átállítja foglaltra, megnöveli *count* értékét, és visszaállítja *mutex1*-et szabadra.
- Ha közben a másik szál szeretné megváltoztatni *count*-ot, előtte a *pthread_mutex_lock()* ellenőrzi, hogy a mutex szabad-e. Ha foglalt, addig vár, amíg fel nem szabadul, és utána ő zárja le.

Vagyis a változóhoz hozzáférést a *pthread_mutex_lock()* függvény gátolja meg.

Join ► Nem szükséges további magyarázat.

Feltétel változók ►

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

pthread_mutex_t count_mutex =
    PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t condition_mutex =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condition_cond =
    PTHREAD_COND_INITIALIZER;

void *functionCount1(void*xxx);
void *functionCount2(void*xxx);
int count = 0;
#define COUNT_DONE 10
#define COUNT_HALT1 3
#define COUNT_HALT2 6

main(){
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, &functionCount1, NULL);
    pthread_create(&thread2, NULL, &functionCount2, NULL);
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    exit(0);
}

void *functionCount1(void*xxx){
    for(;;){
        pthread_mutex_lock( &condition_mutex );
        while( count >= COUNT_HALT1
            && count <= COUNT_HALT2 ){
            pthread_cond_wait( &condition_cond,
                &condition_mutex );
        }
        pthread_mutex_unlock( &condition_mutex );

        pthread_mutex_lock( &count_mutex );
        count++;
        printf("Counter value functionCount1: %d\n",
            count);
        pthread_mutex_unlock( &count_mutex );

        if(count >= COUNT_DONE) return(NULL);
    }
}

void *functionCount2(void*xxx){
    for(;;){
        pthread_mutex_lock( &condition_mutex );
        if( count < COUNT_HALT1
            || count > COUNT_HALT2 ){
            pthread_cond_signal( &condition_cond );
        }
        pthread_mutex_unlock( &condition_mutex );

        pthread_mutex_lock( &count_mutex );
        count++;
        printf("Counter value functionCount2: %d\n",
            count);
        pthread_mutex_unlock( &count_mutex );

        if(count >= COUNT_DONE) return(NULL);
    }
}
```

Míg *count* változó értéke 3 és 6 között van, *functionCount1()* függvény felfüggesztett állapotba kerül. Ha *count* értéke más, *functionCount1()* és *functionCount2()* versenyez egymással, és hol ez, hol az lépteti és írja ki *count* értékét.

A *condition_cond* változót mutexszel kell védeni az összeakadás ellen. A *pthread_cond_wait()* függvény felfüggeszti a saját szálának futását addig, amíg egy másik szál *signal*-t nem küld. Fel is szabadítja a feltétel változót védő mutexet, így nem fagy le a másik szál a várakozás miatt. Miután *functionCount2()* *signal*-t küldött, *pthread_cond_wait()* ismét lefoglalja a feltétel változót védő mutexet, amit az felszabadul (addig vár), és ezután engedni csak tovább a saját szálát.

Ha több szál is várakozik, akkor a *pthread_cond_signal()* csak az egyiket szabadítja föl. Nem meghatározott, hogy melyiket. A *pthread_cond_broadcast()* felszabadítja az összeset.

A többszálúság csapdái ►

- **Versenyhelyzetek:** mutexszel és joinnal megoldható
- **Szálbiztos kód:** kerülni kell a globális és statikus változók használatát. A függvény eredményének a hívó biztosítson helyet!
- **Mutex holtpont (deadlock):** a mutexet szabadítsuk fel, amint lehet
- **Feltétel változó holtpont:** if és while helyes alkalmazásával.

*

Tárgymutató

!

! operátor 14, 51
!= operátor 14, 50

#

#define 94
#elif 94
#else 94
#endif 94
#if 94
#ifdef 94
#ifndef 94
#include 21, 94
#pragma 94
#undef 94

%

% operátor 14, 51

&

& operátor 39, 51
&& operátor 15, 51

*

* operátor 16, 51, 53

,

, operátor *Lásd* vessző operátor

?

? operátor 31, 52

^

^ operátor 39, 51

-

__DATE__ 94
__FILE__ 94
__LINE__ 94
__TIME__ 94
__TIMESTAMP__ 94

|

| operátor 39, 51
|| operátor 15, 51

~

~ operátor 39, 51

<

<< operátor 39, 51

=

= operátor 13, 50
== operátor 14, 50

>

>> operátor 39, 51
>> 75

A, Á

adat 12
adatelem 8
adatszerkezet 8
algorithmus 8
állapotgráf 79
állapottábla 78
álvéletlen szám 98
asctime 98
assembly 45
asszociativitás 50
atof 63
atoi 63
atol 63
auto 68

azonosító 22

B

back slash 13
Backus–Naur Form *Lásd* BNF
bájt 8
bájt mérete 8
balérték 50
belső formátum 43
bináris *Lásd* kettes számrendszer
bináris fa 83
bináris keresés 88
bit 8
Biteltolás 39
bitenkénti
 ÉS *Lásd* & operátor
 KIZÁRÓ VAGY *Lásd* ^ operátor
 NEM *Lásd* ~ operátor
 VAGY *Lásd* | operátor
bitenkénti logikai operátorok 39
bitmező 48, 95
BNF 20
break 29, 31
brute-force módszer 13
bug *Lásd* szemantikai hiba
busz 44

C

C nyelv 9
calloc 58
case 29
case sensitive 22
char 29, 38
ciklus 10
 elől tesztelő 29
 hátral tesztelő 29
 végtelen 30
ciklusmag 10
CISC 45
clock 98
compiler *Lásd* fordító
const 63
const char* 63
continue 31
CPU 45
ctime 98

Cs

csv 72

D

De Morgan azonosságok 15
debugger 17
decimális 13
 kiírás 38
default 29
deklaratív programozási nyelv 12
dokumentálás 10
double 24, 38
 konstans 25

E,É

EBNF 20
egész
 konstans 25
elágazás 10
elemi utasítás 8, 11
élettartam 68
eljárás *Lásd* függvény
előfeldolgozó 20
előjel nélküli egész 8
előjeles egész 8
else 28
enum 47
EOF 42
EPIC 45
értékkadás 13
ÉS
 bitenkénti *Lásd* & operátor
 logikai *Lásd* && operátor
exit 42
extern 68

F

fa 83
fclose 70
fejlécállomány 22
felsorolt típus *Lásd* enum
feltételes operátor 31
fgets 43
FILE * 70
float 38
foglalt szavak 22
folyamatábra 10, 65
fopen 70
for... 26, 29
fordító 17, 20
formátumsztring 13, 43
fő hatás 50
fprintf 42, 71
fputc 71, 72

fputs	71
fread	70
free	58
fseek	70
function	Lásd függvény
funkció	Lásd függvény
függvény	11, 22, 48
definíció	22
deklaráció	22
függvényfej	22
függvénytörzs	22
prototípus	22
függvénykönyvtár	22
függvénypointer	53
fwrite	70

G

gépi kód	8, 20, 45
getchar	42
gets	42
globális változó	67
gmtime	98
goto	32

Gy

gyökér elem	83
--------------------------	----

H

Harvard architektúra	44
használati utasítás	10
háttértár	44
header fájl	Lásd fejlécállomány
hexadeximális	
kiírás	38
hibakezelés	11

I, Í

IDE	17
if	28
igazságtábla	15
imperatív programozási nyelv	12
index	35
információ	12
int	38
konstans	25
isalnum	49
isalpha	49
islower	49

isspace	49
isupper	49
iteráció	Lásd ciklus

J

jobbérték	50
------------------------	----

K

karakter	
konstans	25
karakterisztika	37
kettes komplement számábrázolás	37, 39
kettes számrendszer	8
kezdőérték	14, 33
tömbnek	35
kiértékelési pont	52, 73
kifejezés	23
utasítás	28
kivételkezelés	Lásd hibakezelés
KIZÁRÓ VAGY	Lásd VAGY, kizáró
bitenkénti	Lásd ^ operátor
kódolás	9
konstans	12, 25
kulcsszavak	Lásd foglalt szavak
külső formátum	43

L

láthatóság	68
lebegőpontos	
konstans	25
lebegőpontos típus	24
legkisebb közös többszörös	16
legnagyobb közös osztó	14
lezáró nulla	36
lineáris keresés	88
linker	17, 20
localtime	98
logaritmus keresés	Lásd bináris keresés
logikai	
ÉS	Lásd && operátor
NEM	Lásd ! operátor
VAGY	Lásd operátor
lokális változó	67
lokalitás	68
long	38
long double	38
long long	38
lvalue	Lásd balérték

M

main függvény	22
makró	94
malloc	58
mantissza	37
megjegyzés	21
mellékhatás	50
kiértékelése	51
memória	8, 44
memóriacím	8, 53
memóriaszemét	14
mikroprocesszor	45
módosító	38
mutató	48, 53

N

NEGÁLÁS

bitenkénti	<i>Lásd</i> ~ operátor
négyzetgyök	<i>Lásd</i> sqrt

NEM

bitenkénti	<i>Lásd</i> ~ operátor
logikai	<i>Lásd</i> ! operátor
nemterminális szimbólum	20
Neumann architektúra	44
Neumann elvek	44
NULL	53

Ny

nyíl operátor	75
----------------------------	----

O,Ó

object fájl	20
oktális	
kiírás	38
Operációs rendszer	45
operandus	50
operátor	23
bitművelet	51
feltételes	52, <i>Lásd</i> feltételes operátor
léptető	51
logikai	51
multiplikatív	51
nyíl	75
vessző	52

Ö,Ő

összetett utasítás	11, 28
---------------------------------	--------

P

paraméter	
aktuális	32
formális	32
paramétersor	55
Pascal nyelv	12
perifériák	8
pointer	48, 53
létrehozása	53
NULL	53
típus nélküli (void*)	53
típusa	53
pontosvessző	13
posztdekremens	51
posztinkremens	51
precedencia	24, 50
predekremens	51
preinkremens	51
preprocesszor	<i>Lásd</i> előfeldolgozó
printf	23, 43
processzor	8
program	9
programozás	8
programozási nyelv	8
pszudokód	10
pthread_create	99
pthread_exit	99
pthread_join	99
pthreads	99
putchar	42
puts	42

R

RAM	8, 44
rand	98
realloc	60
register	68
rekurzio	81
kölcsonös	81
önrekurzio	81
remove	98
rename	98
return	22, 31
rewind	70
RISC	45
ROM	44
rövidzár	51, 73
rvalue	<i>Lásd</i> jobbérték

S

scanf	35, 43
--------------------	--------

sequence point	<i>Lásd</i> kiértékelési pont
shiftelés	39
short	38
signed	38
sizeof operátor	59
sorfolytonos	36
specifikáció	9
sprintf	61
sqrt	24
srand	98
sscanf	61
stack	79
static	68
stdarg.h	97
stderr	42, 73
stdin	73
stdio.h	22
stdout	73
stráza	75
strcat	62
strcmp	62
strcpy	62
strlen	62
strncpy	62
strstr	62
struktúra	47
strukturált programozás	10
switch	29

Sz

szabványos	
bemenet.....	34
kimenet.....	34
szekvencia	10
szekvenciális	8
szemantikai hiba	9, 18
szintaktikai hiba	9
szintaxis diagram	20
sztring	29
konstans.....	25
szubrutin	<i>Lásd</i> függvény

T

tárolási egység	28
tárolási osztály	68
terminális szimbólum	20
tervezés	9
tesztelés	9
tevékenységtábla	78

time	98
típus	22
típusjelző	38
típuskonverzió	
explicit	25, 31, 38, 39
implicit	31, 39
típuskonverziós operátor	25, 50
típusos nyelv	13
tolower	49
toupper	49
tömb	35, 47
többszörös	36
töréspont	18
túlsordulás	37

Ty

typedef	48
----------------------	----

U,Ú

union	47, 95
unsigned	38

V

va_arg	97
va_end	97
va_start	97

VAGY

bitenkénti	<i>Lásd</i> operátor
kizáró	15
KIZÁRÓ	<i>Lásd</i> ^ operátor
logikai	<i>Lásd</i> operátor

változó	12
definíció	13, 23
inicializálatlan	14
változó argumentumszámú függvények	97
véletlen szám	98
verem	79
vessző operátor	30
vezérlőkéregek	41
vfprintf	97
VLIW	45
void	22, 33, 38

W

whitespace	15
-------------------------	----

Tematika

1. A számítógép felépítése, programozás, algoritmus, adatszerkezet, a programozás menete, algoritmus megadása pszeudokóddal és folyamatábrával, a teafőző robot példája, adat, változó, konstans, típus, C nyelv, számok kiírása 1-től 10-ig (while), páratlan számok kiírása (if), LNKO, logikai ÉS, VAGY, NEM.
2. A fordítás menete, a szintaxis leírása (BNF, EBNF, szintaxis diagram), a C program részei – mi micsoda az LNKO példáján keresztül, double típus és a másodfokú egyenlet gyökei, precedencia fogalma, konstansok, átlagszámító program, ++, += operátorok, tükörképpével megegyező háromjegyű számok kiírása.
3. Üres utasítás, összetett utasítás, tárolási egység definíciója/deklarációja, kifejezés utasítás, if, else, switch, for, while, do-while, elől és hátul tesztelő ciklusok, ugró utasítások, feltételes operátor, egyszerű függvények felépítése, char kezelése, egész mint logikai érték.
4. Egy- és többdimenziós tömbök, egész és valós típusok fajtái, számábrázolási kérdések, bitműveletek.
5. Bemenet és kimenet: printf, scanf részletesen, getchar, putchar, ASCII, gets, puts. A számítógépek felépítése, működése, a Neumann elvek, operációs rendszerek. Összetett és származtatott adattípusok, enum, struktúra, típusdefiníció. Operátorok áttekintése, balérték-jobbérték, precedencia, asszociativitás, mellékhatás, rövidzár, kiértékelési pont.
6. A számítógép memóriája, pointer fogalma, pointer aritmetika, cím és érték szerinti paraméterátadás, dinamikus memóriakezelés, dinamikus tömb. Sztringek.
7. A programozás menete ismét, struktogram, a C programok felépítése bővebben: több modulból álló programok, globális változók, tárolási osztály, láthatóság, érvényességi tartomány, lokális változók életciklusa. A main() változatai, parancssori paraméterek. Fájlközelés: szöveges és bináris fájlok.
8. Láncolt listák felépítése, a listakezelés algoritmusai, fésűs listák.
9. Az állapotgép fogalma, állapotábra, állapotgráf, megvalósítás C nyelven. A verem fogalma, függvényhívás verem segítségével. Rekúzió fogalma, rekurzív algoritmusok: faktoriális, szám kiírása megadott számrendszerben,
10. Bináris és többágú fák. A fabejárás algoritmusai. Függvénypointer fogalma és használata.
11. Lineáris és bináris keresés. Kiválasztásos, buborék, beszűrő, gyorsrendezés, lista rendezése.
12. A preproceszor: #include, #define és a makrók, #if és a feltételes fordítás, #pragma. Bitmező és union. Változó paraméterlistájú függvények. További hasznos szabványos függvények: véletlenszám, időkezelés, mappaműveletek.
13. Egy komplex program bemutatása, érdekes algoritmusok.