PROGRAMOZÁSI NYELVEK 1. (C)

Juhász István és Pohl László jegyzetei alapján

A számítógép felépítése

- Memória
- Processzor
- Perifériák (I/O)

Alapok

- Kettes számrendszer. Bit, byte
- Intervallumok, előjeles és előjel nélküli számok
- Többféle típusú adattal dolgozhatunk
  - Numerikus
  - Karakteres
  - Logikai, stb
- Számábrázolás egész és valós számok esetében
- Memóriacímzési problémák, módok
- Neumann elv
- Gépi kódú utasítások⇨ Programozási nyelvek

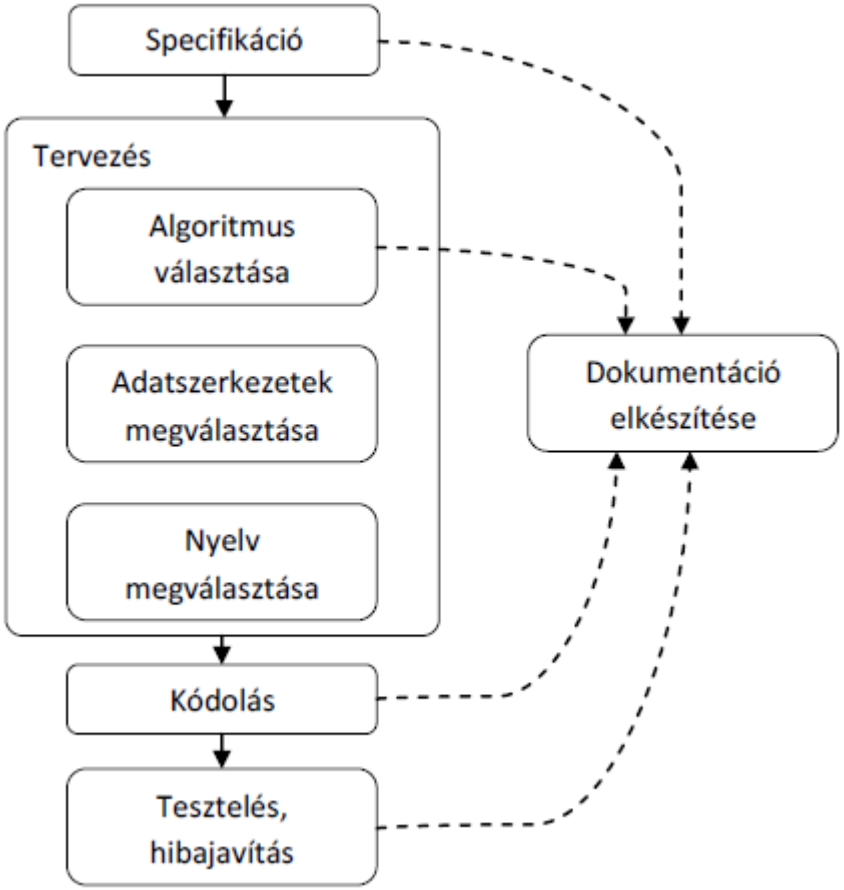
Programozási fogalmak

- Programozás: számítógép-algoritmusok és adatszerkezetek megtervezése és megvalósításuk valamely programozási nyelven.
- Algoritmus (jó és rossz példák)
- Adatszerkezet
- Programozási nyelv

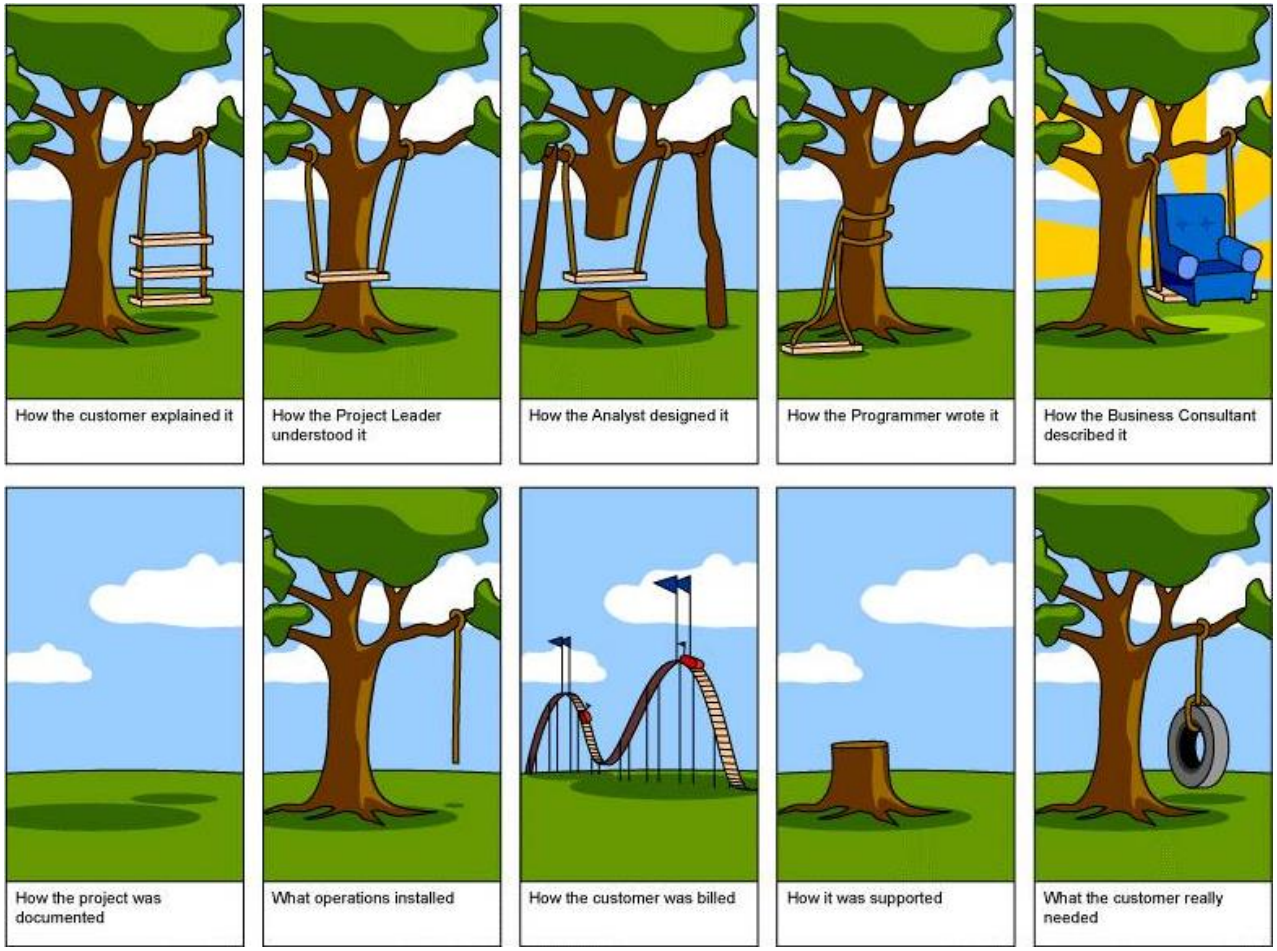
Programozás != kódolás

- Program: algoritmus + adatszerkezet.
- Kódolás: az algoritmus és az adatszerkezet megvalósítása valamely programnyelven.
- Szükséges néhány alapvető programozási algoritmus

A programozás menete



A fejlesztő csapat munkája



Algoritmusok leírása

- Szöveges megadás
  - Pszeudokód
  - Programkód
- Grafikus megadás
  - folyamatábra
  - Struktogram

Példák: http://wiki.prog.hu/wiki/Algoritmus

Példa algoritmusra: teafőzés

- „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.

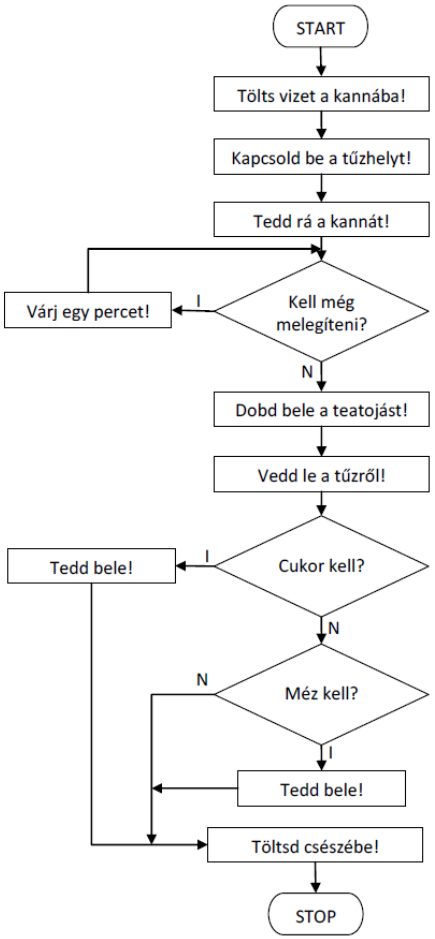
Az algoritmus elemei

- Blokk
- Szekvencia
- Feltételes elágazás
- Iteráció

Egyéb algoritmikus kérdések

- Egyszerű, vagy összetett utasítás?
- Függvények, eljárások (szubrutinok)
- Hibák elhárítása, kivételkezelés

Teafőzés folyamatábrával

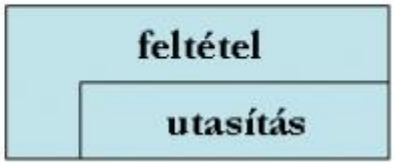


A struktogram elemei

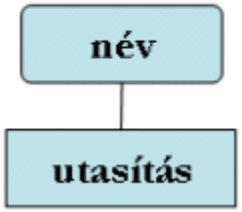
Szekvencia

Elágazás

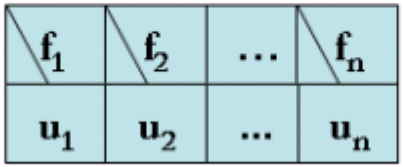
Ciklus



Eljárásdefiníció

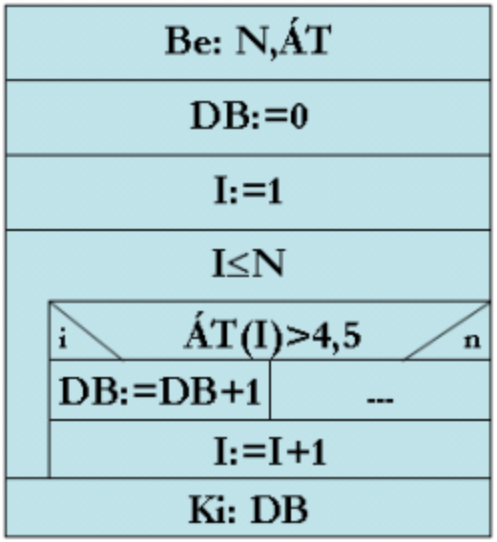


Sokirányú elágazás



Hátultesztelős ciklus

Mit csinál a példa?



(progalap.elte.hu)

Javítsd ki a példát!

- a) Ha nincs bérlet
- b) Lyukaszd!
- c) VÉGE.
- d) Szállj át a következő kocsiba!
- e) Amíg van kezeletlen utas
- f) Kérd a jegyet vagy a bérletet!
- g) Adj pótdíjcsekket!
- h) Ha nem ért a szerelvény végére
- i) Ha van jegy
- j) Amíg nem ér a szerelvény végére
- k) Menj a következő utashoz!
- l) Egyébként

Programozási nyelvek

- Gépi kód, assembly

```
00000000  push    ebp
00000001  mov     ebp, esp
00000003  movzx   ecx, [ebp+arg_0]
00000007  pop     ebp
00000008  movzx   dx, cl
0000000C  lea     eax, [edx+edx]
0000000F  add     eax, edx
00000011  shl     eax, 2
00000014  add     eax, edx
00000016  shr     eax, 8
00000019  sub     cl, al
0000001B  shr     cl, 1
0000001D  add     al, cl
0000001F  shr     al, 5
00000022  movzx   eax, al
00000025  retn
```

- Fortran: tudományos, műszaki nyelv
- Cobol: adatfeldolgozásra, gazdasági számításokra
- Algol60: matematikailag definiált nyelv

Programozási nyelvek #2

- LISP, PROLOG: mesterséges intelligencia támogatására
- BASIC: oktatás mikroszámítógépeken
- PASCAL: általános oktatási célokra
- C: népszerű, a későbbi nyelvekre hat
- C++
- Java: első platformfüggetlen nyelv
- .NET nyelvek (pl. C#)

A nyelvek csoportosítása

- Imperatív nyelvek:
  Utasításszerkezetűek, algoritmikusak
  - Eljárásorientált
    Pascal, C
  - Objektumorientált
    C++, Java, C#
- Deklaratív nyelvek:
  nem az algoritmust, hanem a problémát adjuk meg.
  - Funkcionális
    Lisp
  - Logikai
    Prolog

Újabb programozási alapfogalmak

- Forráskód
- Szintaktika
- Szemantika
- Hivatkozási nyelv
- Implementáció

Programfuttatási módszerek

- Fordítóprogram
  fordító → tárgykód → szerkesztő → futtatható → betöltő
- Interpreter

Előfeldolgozó

- Kódtakarítás
- Fordítási direktívák
- Konstantsok, makrók helyettesítése

Fordító

- Object fájl (gépi kód, vagy asm)

Szerkesztő (linker)

- Összemásolás több fájl esetén
- Szabványos függvényhívások beszerkesztése

A program elemei

karakter → lexikális egység → szintaktikai egység → utasítás → programegység → fordítási egység → program

Karakter

- Karakterkészlet, kódolás (pl. ASCII, UNICODE)
- Típusai:
  - Betűk (ABC, kis-nagybetűre érzékeny?)
  - Számjegyek
  - Egyéb karakterek (speciális jelentés - $, @, # - , szóközök)

Lexikális egységek

- Szimbolikus nevek
  - Azonosító: saját objektumaink megnevezésére. Elnevezési konvenciók, hossz kérdése
  - Kulcsszó: a nyelv számára fenntartva
  - Standard azonosító: nyelv által definiált, a programozó megváltoztathatja
- Elhatároló jelek
  - Kötött formátum: egy sor = egy utasítás
  - Szabad formátum: utasításszeparálás blokkban, utasításvég használata
- Megjegyzés, szerepei (pl. javadoc)
  - Teljes sor
  - Adott szakaszon

Változó

- Név
- Attribútum
- Cím
- Érték

Változók jellemzői

Az attribútum hozzárendelése lehet:

- Explicit
- Implicit
- Automatikus

A címhozzárendelés lehet:

- Abszolút
- Relatív
- Futtatórendszer által megadott

Változók jellemzői #2

Az értékadás történhet:

- Értékadó utasítással (:=, =, iránya, többszörös értékadás, konverzió kérdése)

```
x = y + 3;
x = x * 2;
```

- Input utasítással
- Kezdőérték-adással
  - Explicit
  - Automatikus

Változók (és azonosítók) névadása

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

- Számmal nem kezdődhetnek
- Case Sensitive
- Foglalt szó (kulcsszó: char, int, if) nem adható

Változók deklarálása, felhasználása

```
int eletkor;
float atmero = 4.5;
atmero = atmero + 2;
atmero += 3;
atmero %= 2;
int a;
a++; (!!)
printf(„%d”, a);
```

Nevesített konstans

Programozási eszköz, részei:

- Név
- Típus
- Érték

Nem változtatható (nincs címkomponens, vagy nem elérhető számunkra)

#define max 111

Literál, konstans

Elemei:

- Típus
- Érték

Önmagát definiálja, az alakja dönti el a típusát és az értékét.

C literálok

- Rövid egész
  - Decimális: 23
  - Oktális: 012
  - Hexadecimális: 0x22
- Hosszú egész: 1234L
- Karakter: ‘a’, egésznek számít. Mennyi ‘c’-’a’ ?
- Valós
  - Tizedestört
  - Exponenciális
- Sztring

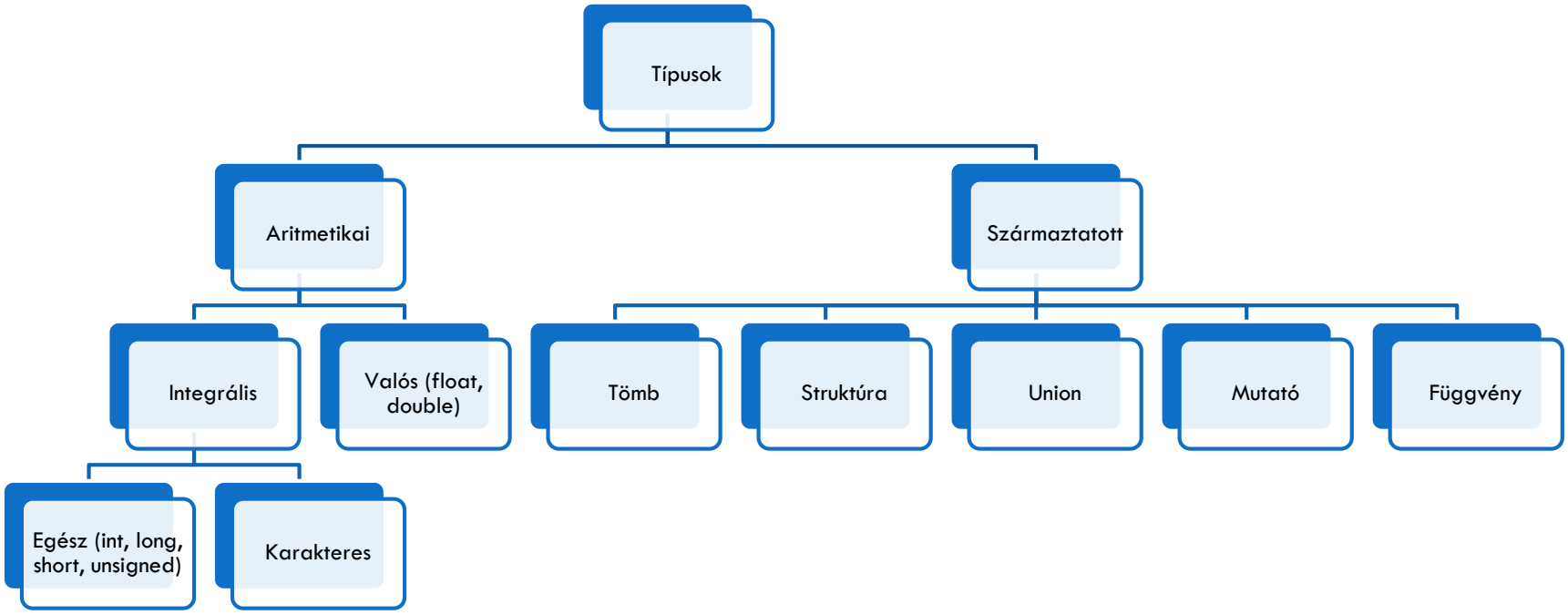
Adattípusok

- Skaláris: atomiak, elemei literálok
- Strukturált: adatszerkezetek megvalósításához

Tömbök, rekordok jelentése

Nyelvfüggő kérdések: a beépített típusok listája, saját típus készíthető-e?

A C típusrendszere



Kifejezések

Szintaktikai eszközök, értékkel és típussal rendelkeznek. Részei:

- Operandusok: literál, konstans, változó, függvényhívás
- Operátorok: műveleti jelek
- Zárójelek: precedencia felülírása

Pl:

```
(sin(x) + 18) * 2.5
2*r*PI > 120
```

Kifejezések operátorai

- Operandus száma szerint:
  - Egyoperandusú
  - Kétoperandusú
  - Háromoperandusú: ? : pl (a>b)?a:b
- Kifejezés alakja szerint:
  - Prefix
  - Infix
  - Postfix

Operátorok

- Előjelek: + -
- Aritmetikai operátorok: + - * / %
- Relációs operátorok: < <= > >= == !=
- Logikai operátorok: ! && ||
- Értékadó operátorok: = += -= *= /= %=
- Bitoperátorok:
  - Eltolás: << >>
  - Bitenkénti logikai: & | ^ (kizáró vagy) ~ (tagadás)

Operátorok, kiértékelési sorrend

Infix alak esetében nem mindig egyértelmű,

ezért precedenciatáblázat segíti:

( ) [ ] . ->	→
* & + - (mint előjel) ! ~ ++ -- sizeof(típus)	←
* / %	→
+ -	→
>> <<	→
< > <= >= == !=	→
& ^ | && || ?:	→
= += -= *= /= %=	←

Mi lesz a változók értéke?

```
int a = 2, b = 3;
a++;
b--;
a += --b;
Vagy: a+= b--;
```

Beolvasás, kiírás

```
printf(”Szöveg\n”);
// Vezérlőjelek:  \n  \t  \”  \\  stb.
int a= 5, b = a+2;
printf(”Első érték: %d, második érték:
  %d”, a, b);
int c;
scanf(”%d”, &c);

//Szöveg beolvasása esetében scanf
  szóközig olvas!
```

Karakterek beolvasása

- Karakter beolvasása
  - `getchar()` : 1 karakter beolvasása
- Szöveg (string) beolvasása
  - `gets()` : beolvasás új sorig
- Formázott beolvasás
  - `char szoveg[100];`
    `scanf("%s", szoveg);` // Itt nincs & !

Mit csinál a program?

```
int a, b, cs;
printf(”a=”); scanf(”%d”, &a);
printf(”b=”); scanf(”%d”, &b);
cs = a; a = b; b = cs;
printf(”a= %d, b = %d”, a, b);
```

Utasítások

Utasítás	Formátum
Üres utasítás	; vagy { }
Összetett utasítás	{ utasítás1; utasítás2;…}
Deklaráció (változók, függvények, konstansok)	típus változó1 [= kezdőérték] [,változó2…];
Kifejezés utasítás	Operátorok, függvényhívások, változók, konstansok sorozata
Feltételes elágazás	if, switch
Ciklus	while, do…while, for
Ugró utasítások	return, break, continue, goto

Üres utasítás

- Szintaktika:

```
;

{   }
```

- Nem csinál semmit, csak szintaktikai okból lehet rá szükség, pl. for ciklus esetében

Összetett utasítás

- Szintaktika:

```
{
  utasítás1;
  utasítás2;
  …
}
```

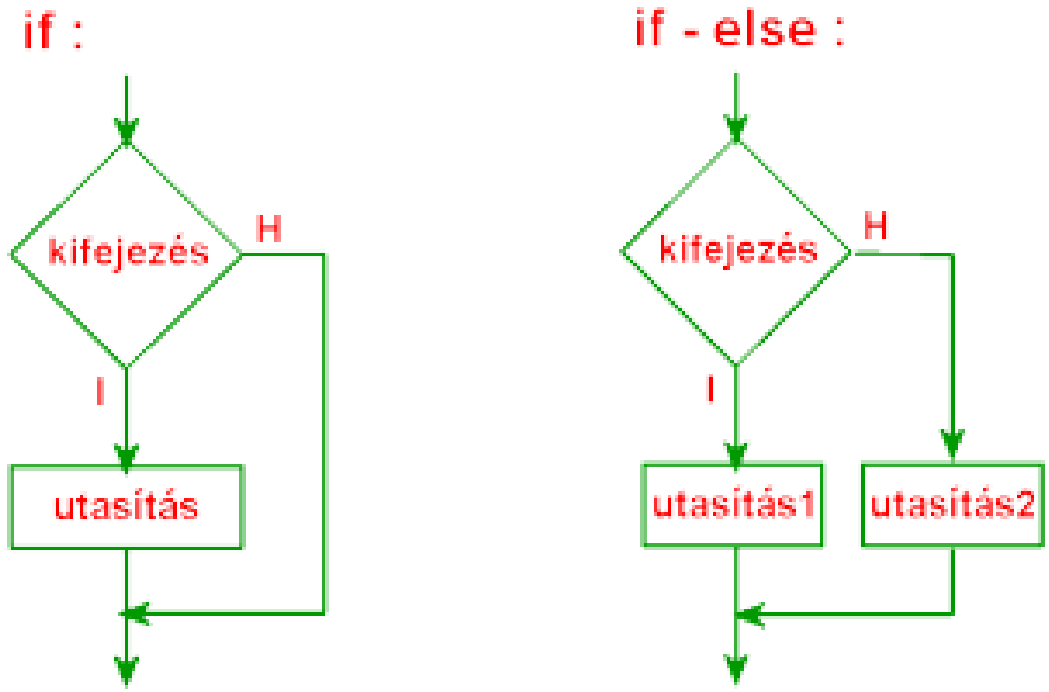
- A behúzás szerepe
- A deklarációk ezen belül érvényesek
- Egymásba ágyazhatók

Összetett utasítás példa

```
{
  printf(”Itt x még nem létezik!”);
  int x=5;
  x += 20;
  printf("%d\n",x);
}
x=10;
// Fordítási hiba: x nem létezik a
  blokkon kívül
```

Feltételes utasítások

Folyamatábra segítségével ábrázolva:



Feltételes utasítások

- Szintaktika:

```
if (feltétel) utasítás;


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

- A feltétel lehet egész kifejezés: ha nem nulla, akkor igaz.
- Az utasítás lehet összetett.

Feltételes utasítások (példa)

```
if ( x < 0 ) printf(”A szám negatív.\n”);
else printf(”A szám nem negatív.\n”);
```

- Hogyan vizsgálhatjuk meg a nulla értéket külön is? Az if utasítások egymásba ágyazhatók:

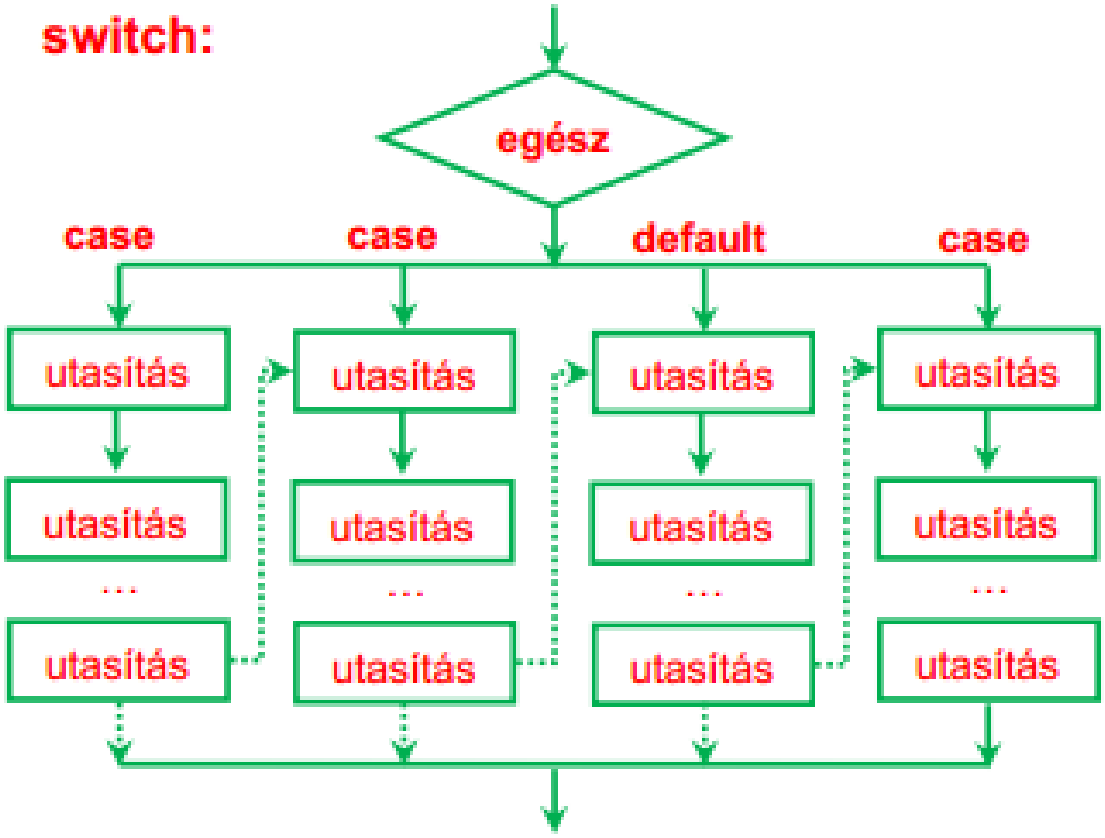
```
if ( x < 0 ) printf(”A szám negatív.\n”);
else if ( x == 0) printf(”A szám
  nulla.\n”);
      else printf(”A szám nem negatív.\n”);
```

Többszörös feltételes elágazás

```
if ( x == 1 ) printf(”Hétfő”);
else if ( x == 2 ) printf(”Kedd”);
else if ( x == 3 ) printf(”Szerda”);
else if ( x == 4 ) printf(”Csütörtök”);
else if ( x == 5 ) printf(”Péntek”);
else if ( x == 6 ) printf(”Szombat”);
else if ( x == 7 ) printf(”Vasárnap”);
else printf(”Hibás x érték!”);
```

Többszörös elágazás (switch)

- Folyamatábrával ábrázolva:



Switch szerkezet

- Szintaktika:

```
switch ( egész kifejezés )
{
  case konstans : utasítás;
  …
  default: utasítás;
}
```

- A default opcionális és bárhol lehet a blokkban.

Switch példa

```
switch (x)
{
  case 1: printf(”Hétfő”);break;
  case 2: printf(”Kedd”);break;
  case 3: printf(”Szerda”);break;
  case 4: printf(”Csütörtök”);break;
  case 5: printf(”Péntek”);break;
  case 6: printf(”Szombat”);break;
  case 7: printf(”Vasárnap”);break;
  default: printf(”Hibás x értéke!”);
}
```

Switch másik példa

```
//Bekérjük a hónap és a nap sorszámát, számítsuk ki: az év hányadik napjára esik?
int honap = 9;
int nap = 28;
int napokszama = 0;
switch(honap)
{
 case 12: napokszama += 30 //november 30 napos;
 case 11: napokszama += 31; //október
 case 10: napokszama += 30; //szeptember
 case 9: napokszama += 31; // augusztus
 case 8: napokszama += 31; // stb.
 case 7: napokszama += 30;
 case 6: napokszama += 31;
 case 5: napokszama += 30;
 case 4: napokszama += 31;
 case 3: napokszama += 28;
 case 2: napokszama += 31;
}
napokszama += nap;
```

Ismétlendő feladatok…

- Gauss feladata: adjuk össze a számokat 1-től 100-ig!
- Hogy néz ki az algoritmusa a feladatnak?
- Hogyan oldotta meg a feladatot számítógép nélkül?

Ciklusok

A ciklusszervező utasítások lehetővé teszik, hogy a program egy adott pontján egy bizonyos tevékenységet akárhányszor megismételjünk.

Általános felépítése:

- fej
- mag
- vég

Az ismétlésre vonatkozó információk vagy a fejben vagy a végben szerepelnek. A mag az ismétlendő végrehajtható utasításokat tartalmazza.

Ciklusok fajtái

- Szélsőséges lépésszámú ciklusok:
  - Üres ciklus
  - Végtelen ciklus (bár ez utóbbi megszakítható utasítással)
- Általános fajtái:
  - Feltételes (kezdő, vagy végfeltétellel)
  - Előírt lépésszámú
  - Felsorolásos

Kezdőfeltételes ciklus

- Szintaktika:

```
while ( feltétel ) utasítás
```

- A feltétel integrális típusú. Ha nem 0, ismétel.
- Az utasítás lehet összetett is.
- Tetszőlegesen egymásba ágyazhatók.

Mit csinál a program?

```
int x;
printf(”x=”);
scanf(”%d”, &x);
while (x > 0)
{
   printf(” * ”);
   x--;
}
```

- Mi történik x = 3, 1, 0, -1 esetén?

Igényel-e ciklust?

- 5 szám bekérése, majd kiíratása a bekéréshez képest fordítva
- Számok bekérése 0-ig, átlag kiíratása
- 100 betű bekérése, magánhangzók / mássalhangzók arányának kiíratása

Mit csinál a pszeudokód?

Legyen A=1!

Amíg A kisebb 10-nél

    Legyen B=0!

    Amíg B kisebb 10-nél

    Írd ki ABA-t!

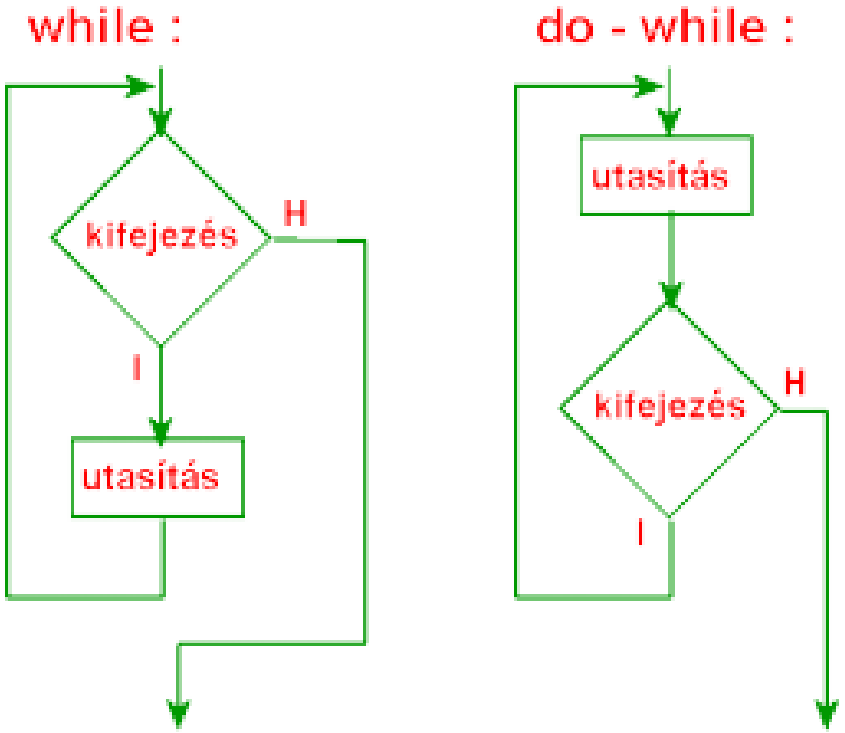
    Növeld B-t eggyel!

Növeld A-t eggyel!

VÉGE.

Kezdő és végfeltételes ciklus

- Folyamatábrával ábrázolva:



Végfeltételes ciklus

- Szintaktika:

```
do
{
  utasítás1;
  utasítás2;
} while ( feltétel );
```

- Addig ismétel, amíg a feltétel nem 0 (mint a while). Legalább egyszer végrehajtja a magot.

Mikor használjunk ilyet?

```
int x;
do
{
  printf(”x pozitív értéke?”);
  scanf(”%d”, &x);
} while ( x <= 0 );
```

Mit csinál a program?

```
int s=0,n;
do
{
  printf(„Adj meg egy szamot:\n");
  scanf("%d",&n); s+=n;
} while(n!=0);
printf(“%d",s);
```

Egymásba ágyazott ciklusok

- A ciklus magja is tartalmazhat ciklust. Ily módon a ciklusok tetszőleges számban egymásba ágyazhatók.

```
int i = 0, j;
while ( i < 3)
{ j = 0;
   while ( j < 2)
  { printf(„%d”, j); j++; }
  i++;
}
```

Lehetséges hibák

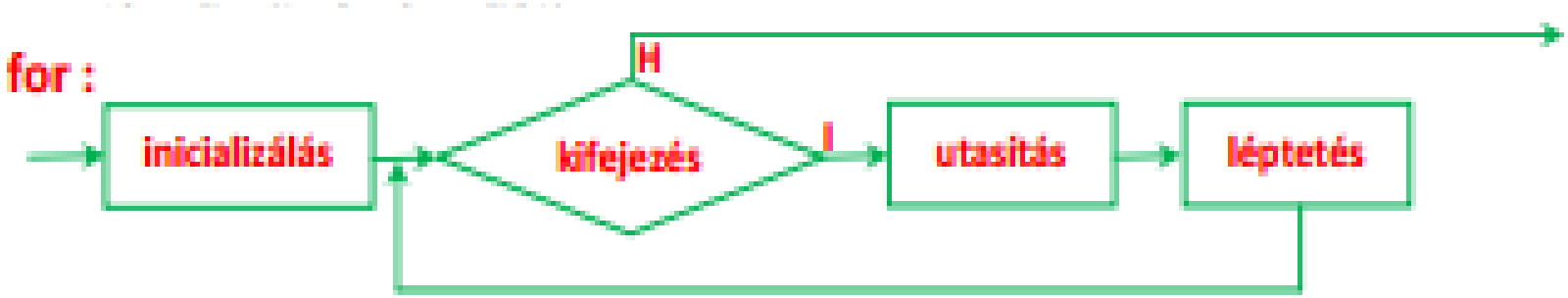
- Ciklusváltozó nem változik
- Beágyazott ciklusban újrafelhasznált ciklusváltozó
- Hatókör-problémák

Egymásba ágyazott ciklusok. Mit csinál a program?

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

A for ciklus

- Folyamatábrán ábrázolva:



For ciklus

- Szintaktika:

```
for ( inicializálás; kifejezés;
  léptetés ) utasítás;
```

- Példa:

```
int i;
for (i = 0; i < 5; i++) printf(”*”);
```

- A for fejének bármely része elhagyható (ha a középső kifejezést hagyjuk el, örökké igaz lesz), vagy összetett kifejezést is írhatunk, vesszővel elválasztva őket.

Mit csinál a kódrészlet?

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

Mit csinál a program?

```
int i, j;
for (i = 0; i < 5; i++)
{
    for (j = 0; j <= i; j++)
    printf(”*”);
  printf(”\n”);
}
```

Feltétel nélküli vezérlésátadás

- Szintaktika:

```
break;
```

- Félbeszakítja a ciklust, vagy a switch-et.

- Szintaktika:

```
continue;
```

- Félbeszakítja a ciklusmagot, és folytatja a ciklust (a feltételkiértékelést, vagy léptetést).

Mit csinál a kódrészlet?

```
int i = 10;
while ( i > 0 )
{
  printf("Hello %d\n", i );
  i = i -1;
  if( i == 6 ) { break; }
}
```

Feltétel nélküli vezérlésátadás

- Szintaktika:

```
goto címke;
```

- A címkével jelzett utasításon folytatja a végrehajtást. Indokolt esetben használjuk, pl. ha egymásba ágyazott ciklusokból, vagy switch-ekből szeretnénk kilépni.

Példa a goto használatára

```
int d;
while (1) {
  scanf("%d", &d);
  switch (d) {
  case 1: printf("Elégtelen."); break;
  case 2: printf("Elégséges."); break;
  case 3: printf("Közepes."); break;
  case 4: printf("Jó."); break;
  case 5: printf("Jeles."); break;
  default: printf("Nincs ilyen osztályzat.");
  goto ciklusvege; } }
ciklusvege:
//Program vége
```

Gyakorlás

- Írasd ki az első 10, n-nel osztható számot!
- Készíts programot, ami egy bekért számról eldönti, hogy tökéletes szám-e!
- Írasd ki n-ig a tökéletes számokat!

Tömbök

Nagyobb mennyiségű adat feldolgozására.
A tömb olyan összetett változó típus, amely a programozó által kívánt mennyiségű, azonos típusú adatot képes tárolni.

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

Tömbindex: az elemek elérésére az index szolgál. A számozás 0-tól kezdődik.

Mikor lehet szükségünk tömbre?

- Bekért szövegről megállapítani, hogy palindrom-e!
- Kérjük be az ötös lottó számait, majd írassuk ki növekvő rendben őket!
- Kérjünk be 10 számot és írassuk ki a legnagyobbat közülük!
- Kérjünk be 50 számot és írassuk ki visszafelé őket!
- Kérjünk be 10 számot és írassuk ki az átlagukat!
- Kérjünk be 10 számot és írassuk ki az átlag felettieket!

Tömbök #2

A tömb méretét kötelező megadni. A tömb nem másolható az = operátorral (tomb1 = tomb2), elemenként kell elvégezni a másolást.

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

Kezdőérték adható a tömbnek:

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

Ilyenkor adhatunk kevesebb értéket, mint a tömb mérete, vagy meg sem kell adni a tömb méretét.

Tömbök deklarálása

```
//Mit ír ki?
int tomb[5];
printf(„%d”, tomb[0]);
int masik_tomb[5] = { 10, 20, 30, 40, 50 };
printf(„%d”, masik_tomb[1]);
int harmadik_tomb[5] = { 10, 20, 30, 40};
//Implementációfüggő eredmény
printf(„%d”, harmadik_tomb[4]);
int negyedik_tomb[] = { 10, 20, 30, 40, 50 };
printf(„%d”, negyedik_tomb[4]);
```

Tömbök méretlekérése

`int a[10];`

`sizeof(a)` – tömb számára lefoglalt memória

`sizeof(a[0])` – egyetlen elem mérete

`int n=sizeof(a)/sizeof(a[0]);`

a tömb elemeinek a száma

Mit csinál a kódrészlet?

```
int tomb[100];
int i, n = sizeof(tomb)/sizeof(int);
for (i = 0; i < n; tomb[i++] = 0);
//Mit ismétel a ciklus?
```

Tömb beolvasása, kiírása

```
int a [10], i;
for (i=0; i<10; i++)
  scanf("%d", &a[i]);
for (i=0; i<10; i++)
  printf(" %d ", a[i]);
```

Mit ír ki a program?

```
#define N 50;
int main()
{
  int i, a[N];
  for (i=0; i<N; i++)
     printf(" %d ", a[i]);
}
```

Szöveg, mint tömb

A string, vagy karakterlánc karakterek sorozata.

```
char szoveg[100];
szoveg[0] = 'a';
szoveg[1] = 'l';
szoveg[2] = 'm';
szoveg[3] = 'a';
```

Szöveg, mint tömb

A stringet az End Of String jel zárja le! (\0)

```
char szo1[256];
```

szo1-ben legfeljebb 255 hosszú karakterláncok tárolhatók. Az elemek értéke határozatlan.

```
char szo2[10] = {'a', 'l', 'm', 'a', '\0'};
```

szo2 az "alma" karakterláncot tárolja. A maradék elemek értéke határozatlan.

```
char szo3[10] = "alma";
```

Ekvivalens az előzővel, de tömörebb, olvashatóbb. Nem a tömb kap értéket, hanem az elemei!

```
char szo4[10] = {'a', 'l', 'm', 'a'};
```

Nem sztringet tárol, hacsak az adott implementáció ki nem nullázza a tömb maradék elemeit.

Szöveg beolvasása

```
char szoveg[100];
scanf("%s", szoveg);
```

Kivételesen nem írunk & jelet a scanf-be!

A beolvasás csak az első whitespace-ig tart!

Különben használjuk a `gets()` függvényt!

Bemutatás példával.

Probléma: szám beolvasása után karakter bevitele.

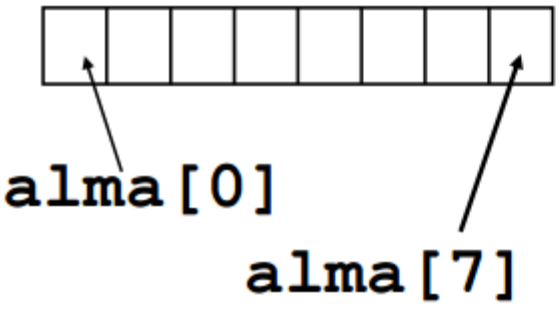
Mit olvas be a program?

```
int szam;
scanf("%d", &szam);
getchar(); /* A getchar az előbbi
szám után beolvasott enter-t tünteti
el. */
char betu;
scanf("%c", &betu);
```

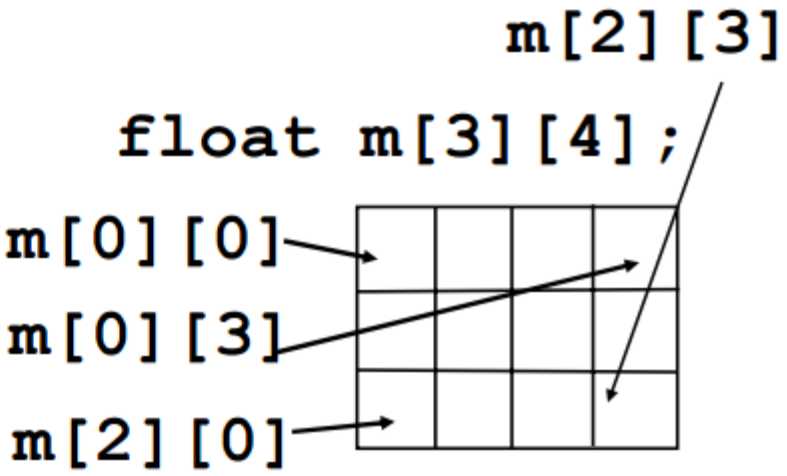
Tömb típusú tömbök

Pl:

```
int alma[8];
```




```
float m[3][4];
```



Többdimenziós tömbök

C-ben nincs valódi többdimenziós tömb, de ez helyettesíthető tömb típusú tömbökkel, amelyek sorfolytonosan tárolódnak.

```
int tomb[3][3];
int i, j;
for (i = 0; i < 3; i++)
     for (j = 0; j < 3; j++)
          {tomb[i][j]= i*3+j;}
//Írassuk ki a fenti tömb elemeit!
```

Többdimenziós tömb feltöltése deklaráláskor

```
//Nem szükséges a sorok száma!
int tomb[][3]= {{0,1,2},
                {3,4,5},
                {6,7,8}};
//Mi történik, ha valamelyik sor
  hiányos, vagy több elemet
  tartalmaz?
```

Feladat tömbökkel

Készíts tömböket IP címek tárolására! Kérj be egy saját és egy távoli IP címet, valamint a saját címhez tartozó maszkot! Állapítsd meg a hálózat címét, továbbá azt, hogy kell-e átjárót igénybe venni a távoli cím elérésére! Opcionális: broadcast cím kiszámítása. Használjuk a bitenkénti & operátort!

Saját típusok

A `typedef` segítségével új típusneveket készíthetünk a következő szintaxissal:

```
typedef eredeti_típus új_típusnév;
```

Példák:

```
typedef int egesz;
egesz a = 5;
typedef egesz tomb[100];
tomb t1, t2;
```

Felsorolás típus

Absztrakciós típus, aminél az értékkészlet előre megadott értékek halmaza.

```
enum lampa_szine { piros, sarga,
zold };
enum lampa_szine lampa = sarga;
```

Egész konstansokat tárol az értékek mögött. Ezek értéke megadható, vagy automatikusan növekszik, alapesetben 0-tól.

Felsorolás típus saját értékkel

```
enum kartyalapok {also = 2, felso,
kiraly, hetes=7, nyolcas, kilences,
tizes, asz};

enum kartyalapok kartyalap;
kartyalap = kiraly;
printf("%d", kartyalap);
```

Felsorolás típus több értékkel

A felsorolástípus megfelelő értékválasztásával egy változó több értéket is tárolhat. Az értékek különböző helyiértékű biteket jelöljenek. Több érték között ilyen módon a bitek közötti művelettel (&, |) lehet választani.

Felsorolás több értékkel, példa

```
enum het_napja {hetfo=1,kedd=2,szerda=4};
enum het_napja talalkozo={hetfo|szerda};

//Mennyi lesz a változó értéke?
printf("%d", talalkozo);

if (talalkozo & hetfo) printf("Hetfo");
if (talalkozo & kedd) printf("Kedd");
if (talalkozo & szerda) printf("Szerda");
```

Feladat felsorolásra

Készíts programot, ami egy tömbbe begyűjti felsorolással, hogy a cég munkatársainak mely napok jók egy közös találkozóra! A program írja ki, hogy melyek azok a napok, amelyek mindenkinek megfelelőek lehetnek!

Feladat: „ly” számláló

Készítsünk programot, ami egy szövegben megszámolja az „ly” betűk számát! Az „lly” kettőnek számít!

A megoldáshoz ismerni kell az aktuális betű „előzményeit”. Mik az előzmények?

- Volt-e „l” betű?
- Ha volt, akkor hány „l” betű volt?

Hogyan emlékezhetünk az előzményekre?

- Kóddal, azaz hol jár a program.
- Adattal, vagyis változóban tároljuk.

Programozási tételek

- Általánosított algoritmusok, gyakran ismétlődő tevékenységek megoldására
- A megoldásuk helyessége matematikai úton bizonyított
- Biztonságot ad a használatuk, bár a megfelelő tétel felismerése a konkrét feladatból némi gyakorlottságot igényel

Tételek csoportosítása

- Egyszerű tételek: egy bemenő sorozatból egyetlen érték lesz az eredmény
- Összetett tételek: egy, vagy több sorozatból egy, vagy több sorozat az eredmény

Egyszerű programozási tételek

- Sorozatszámítás, összegzés
- Eldöntés
- Kiválasztás
- Megszámlálás
- Szélsőérték-keresés
  - Minimumkeresés
  - Maximumkeresés
- Keresés
  - Lineáris
  - Logaritmikus (intervallum-felezéses)

Összetett programozási tételek

- Kiválogatás
- Szétválogatás, helyben szétválogatás
- Összefésülés
- Rendezések
- Halmazműveletek
  - Metszet
  - Unió
  - Különbség

Sorozatszámítás, összegzés

Példák:

- Az elmúlt hónapban minden nap futottam különböző távokat. Hány kilométert tettem meg összesen?
- Számoljuk ki a tanulmányi átlagunkat a jegyek ismeretében!

Általános megfogalmazása

Hogyan csináljuk?

Sorozatszámítás, összegzés #2

```
Eljárás:
    S:=0
    Ciklus I=1-től N-ig
        S:=S+A(I)
    Ciklus vége
Eljárás vége.
```

Eldöntés

Példák:

- Döntsük el a jegyek alapján, hogy kitűnő-e a tanuló, vagy sem!
- Döntsük el, hogy egy kapott mondatban szerepelnek-e ékezetes betűk, vagy sem!

Miben különbözik az előző tételtől?

Hogy néz ki az általános alakja?

Eldöntés #2

```
Eljárás:
        I:=1
                Ciklus amíg I<=N és A(I) nem T típusú
                        I:=I+1
                Ciklus vége
        VAN:=I<=N
Eljárás vége
```

Kiválasztás

Példák:

- Hol szerepel egy szövegben először felkiáltójel?
- Hányadik eladott autó volt egy kereskedésben először drágább, mint 4 millió Ft?

Általános alakja, algoritmus megfogalmazása szóban

Kiválasztás #2

```
Eljárás:
    I:=1
    Ciklus amíg A(I) nem T típusú
        I:=I+1
    Ciklus vége
    SORSZ:=I
Eljárás vége
```

Megszámlálás

Példák:

- Hány negatív száma van egy sorozatnak?
- Hányszor futottam a múlt hónapban 10 km felett?
- Hány bukás volt a tanulók között?

Hogyan működik? Miben különbözik az előző tételektől?

Megszámlálás #2

```
Eljárás:
      S:=0
      Ciklus I=1-től N-ig
            Ha A(I) T típusú akkor S:=S+1
      Ciklus vége
Eljárás vége.
```

Szélsőérték-keresés

Példa:

- Egy repülőről rendszeresen megmérjük az alattunk lévő táj tengerszint-feletti magasságát. Ha a tenger felett repülünk, 0 métert mérünk, különben pozitív számok sorozatát kapjuk. Hány méter volt a legmagasabb pont, ami felett elrepültünk?

Milyen előfeltevést tehetünk a sorozatra nézve? Mennyiben kell módosítani ezt akkor, ha a feladat nem tartalmaz ilyet?

Szélsőérték-keresés #2

```
Eljárás:
      INDEX:=1
      Ciklus I=2-től N-ig
            Ha A(INDEX) < A(I) akkor INDEX:=I
      Ciklus vége
Eljárás vége.
```

Ez melyik szélsőérték helyét jelöli ki? Hogyan járjunk el a másik esetben? Mi a teendő, ha magát a szélsőértéket is tárolni kívánjuk?

Lineáris keresés

Példa:

- Van-e Béla keresztnevű hallgató a csoportban, s ha igen, hol? Nem biztos, hogy van, ahogy az sem, hogy csak egy van. Mindenesetre most megelégedünk az első találattal (ha van egyáltalán). A névsorunk nem rendezett – legalábbis a keresztnevekre nem.

Miben hasonlít a kiválasztás tételéhez, illetve miben különbözik attól?

Lineáris keresés #2

Eljárás:

```
I:=1
Ciklus amíg I<=N és A(I) nem T tulajdonságú
        I:=I+1
Ciklus vége
VAN:=I<=N
Ha VAN akkor SORSZ:=I
```

Eljárás vége.

Lineáris keresés strázsával

Hogyan lehet gyorsítani a lineáris keresést, ha a keresendő sorozat nem rendezett a keresési feltételre nézve? Hogyan javít a strázsa-technika a lineáris keresésen? Rövidzár-kiértékelés kérdése a lineáris keresésnél.

Logaritmikus keresés

Példa:

- Hogyan lehet a „Gondoltam egy számra 1 és 100 között, találd ki, hogy melyikre?” játékot hatékonyan játszani? Mennyiben speciális ez a példa?

Hogyan néz ki az algoritmus általános esetben? Milyen feltétel szükséges a sorozatra nézve a kereshetőség érdekében? Miért hívják logaritmikusnak az algoritmust?

Logaritmikus keresés #2

```
Eljárás:
        A:=1
        F:=N
        Ciklus
                K:=INT((A+F)/2)
                Ha T(K)<X akkor A:=K+1
                Ha T(K)>X akkor F:=K-1
        amíg A<=F és A(K)!=X
        Ciklus vége
        VAN:=Al<=F
        Ha VAN akkor SORSZ:=K
Eljárás vége.
```

(Hogyan gyorsíthatjuk egy rendezett sorozatban még tovább a keresést?)

Kiválogatás

Példa:

- Adott egy tömbben a csoport hallgatóinak félévi jegye. Gyűjtsük ki a bukott tanulók sorszámait egy külön tömbbe!

Az algoritmus „összetett”, vagyis nem egyetlen eredményt képes szolgáltatni, hanem eredmények egy sorozatát is.

Kiválogatás #2

```
Eljárás:
      J:=0
      Ciklus I=1-től N-ig
            Ha A(I) T típusú, akkor J:=J+1,
            B(J):=I
      Ciklus vége
Eljárás vége.
```

Mekkora legyen legfeljebb a B tömb?

Szétválogatás

Példa:

- Az előző példa továbbgondolása: mind a bukott, mind a megfelelt tanulók sorszámait gyűjtsük bele egy-egy tömbbe! Ezúttal tehát nincs kimaradó tanuló.

Szétválogatás #2

```
Eljárás:
        J:=0, K:=0
        Ciklus I=1-től N-ig
                Ha A(I) T típusú, akkor J:=J+1, B(J):=I
                Különben K:=K+1, C(K):=I
        Ciklus vége
Eljárás vége.
```

Mekkora legyen legfeljebb a B, illetve a C tömb? Mi a következménye ennek az algoritmusra nézve? Mi lehet a javítása?

Helyben szétválogatás

Módosítsuk úgy az előző algoritmusunkat, hogy egyetlen tömb két végébe gyűjtsük a kétféle tulajdonságú elemeket a sorozatból! Mi akadályozza meg az elemek összemosódását középen?

Összefésülés (összefuttatás)

Példa:

- Adott két, rendezett e-mail címlista. Egyesítsük őket egyetlen listában úgy, hogy a közös címek csak egyszer szerepeljenek!

Használjuk ki a rendezettséget a listákban, ne halmazszerűen próbáljuk megoldani a feladatot!

Összefésülés #2

```
Eljárás:
        I:=1
        J:=1
        K:=0
        Ciklus amíg I<=N és J<=M
                K:=K+1
                Elágazás
                        A(I) < B(J) esetén C(K):=A(I)
                        I:=I+1
                        A(I) = B(J) esetén C(K):=A(I)
                        I:=I+1
                        J:=J+1
                        A(I) > B(J) esetén C(K):=B(J)
                        J:=J+1
                Elágazás vége
        Ciklus vége
```

Összefésülés #2 (folytatás)

```
        Ciklus amíg I<=N
                K:=K+1
                C(K):=A(I)
                I:=I+1
        Ciklus vége
        Ciklus amíg J<=M
                K:=K+1
                C(K):=B(J)
                J:=J+1
        Ciklus vége
Eljárás vége.
```

Rendezések

A rendezési eljárások hatékonysága összehasonlíthatóak a következő szempontok szerint:

- Végrehajtási idő
- Igényelt tárterület
- Végzett műveletek száma (összehasonlítás, csere)

Az egyik legegyszerűbb, cserébe legrosszabb hatékonyságú módszer a teljes cserés rendezés

Teljes cserés rendezés

```
Eljárás:
     Ciklus I=1-től N-1-ig
          Ciklus J=I+1-től N-ig
               Ha A(J) < A(I) akkor C:=A(J)
               A(J):=A(I)
               A(I):=C
          Ciklus vége
     Ciklus vége
Eljárás vége.
```

Halmazműveletek

Ezekben az algoritmusokban a sorozatokat rendezetlennek, halmazszerűnek tekintjük. Feltételezzük ennek folyományaként azt is, hogy egyes elemek nem szerepelnek ugyanazon a sorozaton belül többször is.

Metszetképzés

```
Eljárás:
      CN:=0
      Ciklus I=1-től N-ig
            J:=1
            Ciklus amíg J<=M és A(I)<>B(J)
                  J:=J+1
            Ciklus vége
            Ha J<=M akkor CN:=CN+1
            C(CN):=A(I)
      Ciklus vége
Eljárás vége.
```

Unióképzés

```
Eljárás:
        Ciklus I=1-től N-ig
                C(I):=A(I)
        Ciklus vége
        CN:=N
        Ciklus J=1-től M-ig
                I:=1
                Ciklus amíg I<=N és A(I)<>B(J)
                        I:=I+1
                Ciklus vége
                Ha I>N akkor CN:=CN+1
                C(CN):=B(J)
        Ciklus vége
Eljárás vége.
```

Struktúra

- A struktúra speciális összetett típus, különböző típusú elemeket tudunk összefogni vele. Pl:

```
struct felhasznalo
{
  int eletkor;
  char nev[30];
};
```

A struktúrában lévő `eletkor` és `nev` változók a struktúra mezői.

A struktúra használata

- Innentől kezdve ebből az új típusból tudunk változót deklarálni:

```
struct felhasznalo a;
a.eletkor=18;
strcpy(a.nev,"Géza");
```

- A struktúrában lévő értékekre a minősítő (.) operátorral tudok hivatkozni. A struktúrák között működik az = operátor.

Struktúra másolása

```
struct koordinata
{
  int x;
  int y;
};
struct koordinata a,b;
b.x = 11; b.y = -5;
a=b;
```

Struktúrák tömbben

- A struktúrákat tömbbe is szervezhetjük.

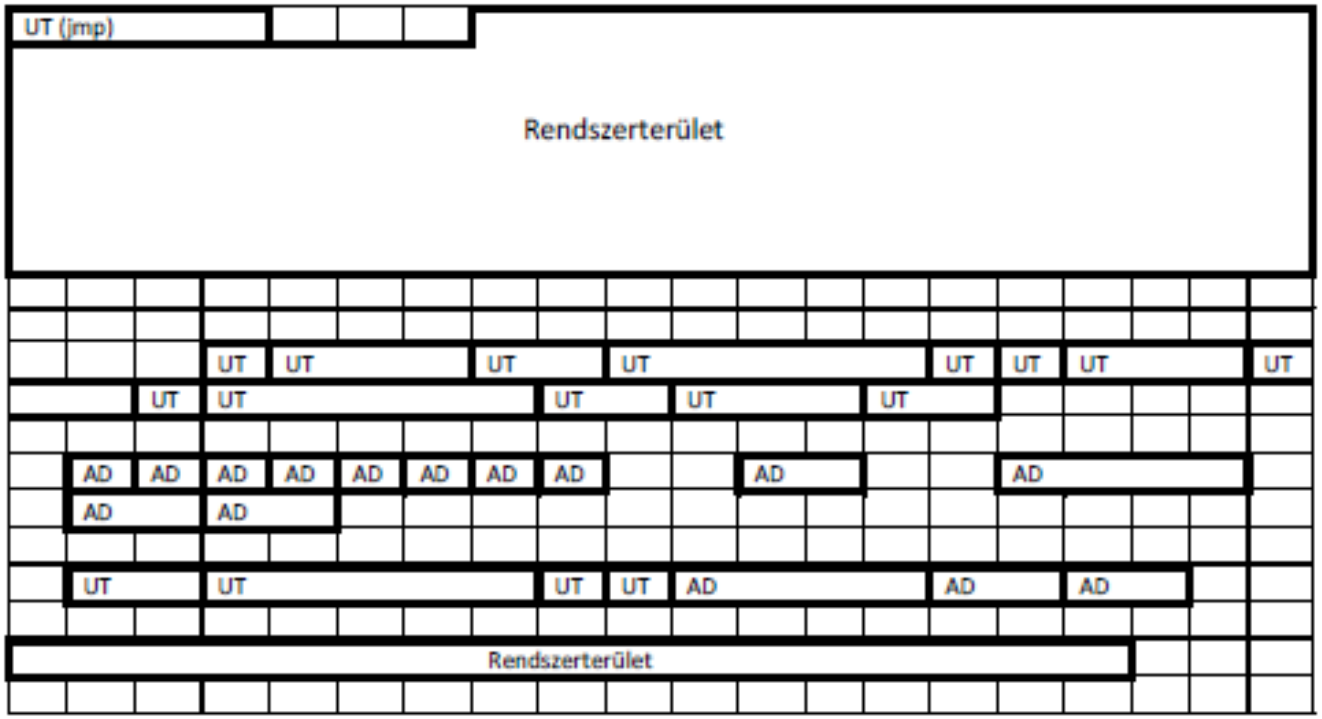
```
struct felhasznalo diak[5];
diak[0].eletkor=20;
strcpy(diak[0].nev, "Béla");
printf("Az 1. diák neve: %s,
  életkora: %d",
diak[0].nev, diak[0].eletkor);
```

Struktúra, mint típus

- Ha hosszúnak találjuk mindig kiírni a `struct felhasznalo` típusnevet, a `typedef` kulcsszóval saját típusnevet hozhatunk létre, melyet azután használhatunk is:

```
typdef struct felhasznalo user;
user diak;
```

A számítógép memóriája



A pointer (mutató)

- A **memóriacím** egy egész szám, amely kijelöli a memória valahányadik bájtját.
- A **pointer** (mutató) olyan változó, amelyik memóriacímet tartalmaz.
- **Pointer típusa**: amilyen típusú adatra vagy függvényre mutat a pointer.
- A pointer **létrehozása** az indirekció operátorral (*) történik.

Pointer létrehozása, void, NULL

- Int-re mutató típusos pointer létrehozása:

```
int * p;
```

A * bárhová írható: `int* p;` `int *p;` `int*p;`

- Típus nélküli pointer: `void * p;`
- NULL pointer: a memória 0. bájtjára mutat. Egy pointer értékét akkor állítjuk NULL-ra, ha jelezni akarjuk, hogy a pointer nincs beállítva.

Változó memóriacíme

- A változó nevéből a címképző (&) operátorral kaphatjuk meg a változó címét:

```
int x,*p;
p=&x;
```

- Pointer által mutatott érték: ha p a pointer, *p az általa mutatott értéket jelenti.

```
int n,*p;
p=&n;
// Az alábbi két értékadás ugyanaz
n=8;
*p=8;
```

Kezdőértékadás

```
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.
- Hogyan változtathatjuk meg az n értékét a p segítségével?

Pointer és tömb

- Egy tömb neve az index operátor ([]) nélkül egy pointer a tömb első (azaz 0 indexű) elemére mutat.

```
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];
```

Tömbmanipulálás pointerrel

```
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;
```

Túlhivatkozás

- Ha olyan memóriát akarunk használni, ahol nincs ténylegesen adat, futásidejű programhibát kapunk!

```
int t[100],*p=t,n;
p=&n;
p[0]=8;
p[1]=8;
```

Mi a hiba a fenti kódban? Miért?

Pointeraritmetika

```
int t[20],*p=t+10;
*(p-2)=6; // azaz t[8]=6;
*(p+3)=6; // azaz t[13]=6; A hozzáadott
  szám mennyivel növeli a memóriacím
  értékét?


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;
```

Pointeraritmetika

- Mennyi lesz az eredmény?

```
int t[20],*p,*q;
p=t+3, q=t+17;
printf("q-p==%d, t-p=%d\n",q-p,t-p);
```

Függvények

- A programok részekre bontásának eszközei. Céljuk:
  - A többször használt programrészeket csak egyszer kell lekódolni.
  - A bonyolult algoritmusok szétbontva egyszerű részekre áttekinthetővé válnak.
- A függvényt tekinthetjük tehát egy adatfeldolgozó egységnek, amely kap bemenő adatokat, és ezeket felhasználva produkál valamilyen eredményt.
- A függvény meghívásának leírását prototípusnak nevezzük.

Függvény paramétere, szerkezete

- A C nyelvben egy függvény tetszőleges számú bemenő paraméterrel és egy darab visszatérési értékkel rendelkezhet. Ha egynél több visszatérés értéket kell adni, azt trükkel oldjuk meg.
- Felépítése:

```
típus függvénynév (formális paraméterlista)
{
  utasítások;
  return visszatérési_érték;
}
```

Példa

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

A függvény meghívása közben paraméterátadás történik:

```
x=6;
y=8;
z=osszead(x,y); //x és y az aktuális
  paraméterek
```

Lokális változók, void függvények

- Az egyik függvényben definiált változó a másik függvényben nem látható. Az esetleg ugyanolyan néven definiált változó máshová mutat.
- Azaz egy függvényben deklarált változó lokális lesz a függvényre nézve.
- Azok a függvények, amelyek nem adnak vissza értéket, `void` típussal rendelkeznek. Ezeknél a `return` kulcsszó is hiányozni fog. Hívásuk történhet utasításszerűen (eljárás).

Mi történik a következő példában?

```
void cserel(int a,int b)
{
  int c;
  c=a;
  a=b;
  b=c;
}
```

Mi történik, amikor meghívom?

Paraméterátadás

- Az eddigi példákban érték szerinti paraméterátadás történt.
- A cserel függvény helyes működéséhez cím szerinti paraméterátadásra van szükség.
  Cím szerinti paraméterátadásnál nem a változó értékét, hanem annak memóriacímét adjuk át a függvény paramétereinek.

Cím szerinti paraméterátadás

```
void cserel(int *a, int *b)
{
  int c;
  c=*a;
  *a=*b;
  b=*c;
}
```

Híváskor:

cserel(**&x,&y**);

Mi történik, ha híváskor nem írunk & jelet?

Tömb átadása paraméterként

Egydimenziós tömb paraméterként való átadása egy függvénynek többféle módon is történhet:

- Pointerrel:
```
void myFunction(int *param)
```
- Adott méretű tömbbel:
```
void myFunction(int param[10])
```
- Méret nélküli tömbbel:
```
void myFunction(int param[])
```

Tömb visszaadása eredményként

- Tömb visszaadása eredményként egy függvényből mutató segítségével lehetséges:

```
int * myFunction() {
```

- A visszatérési érték maga a tömb lesz (pontosabban annak kezdőcíme).

Tömb visszaadása: példa

```
int * TombVissza( ) {
   static int  r[10];
   int i;
   for ( i = 0; i < 10; ++i)  r[i] = i;
   return r;
}
int main () {
int *p;
   int i;
   p = TombVissza();
   for ( i = 0; i < 10; i++ ) {
      printf( "*(p + %d) : %d\n", i, *(p + i));
   }
   return 0;
}
```

Gyakorló feladatok

- Készítsünk függvényt, ami egy paraméterként kapott egész tömb átlagát adja vissza eredményül!

- Készítsünk függvényt, ami a paraméteréül kapott egész számból (0<x<=10) egy latin négyzetet rajzol a képernyőre! Példa x = 5-re:

1 2 3 4 5

2 3 4 5 1

3 4 5 1 2

4 5 1 2 3

5 1 2 3 4

Gyakorló feladatok

- Készítsünk függvényt, ami egy henger térfogatát kiszámítja! A függvény ne adjon vissza értéket! A függvény prototípusa:

```
void terfogat(double d, double h,
  double *terf);
```

Hatáskör

- Megadja, hogy a deklaráció a forráskód mely részeiben használható fel. Alapesetben a deklaráció helyétől a blokk végéig tart.
- Mi történik, ha egy változót hatáskörön kívül igyekszünk felhasználni? Próbáljuk ki!

Hatáskörök célja

- A program logikailag kisebb részekre darabolása (pl. függvényekkel), majd azok adatcseréjének biztosítása. Egyes adatokra mindenütt szükség van, másokat pl. a biztonság okán rejtsünk el.
- Alapesetei:
  - Függvényben, vagy blokkon belül: lokális változó
  - A függvényeken kívül: globális változó
  - Függvény paramétereként megadott: formális paraméter

Újradeklarálás

- Készíthetünk-e egy változó hatáskörén belül egy másik, ugyanolyan nevű változót?
- Készíthetünk-e egy új, azonos nevű változót a változó után egy új blokkban?
- Készíthetünk-e lokális változót, egy globális változóval azonos névvel? Elérhetjük-e a globális változót ekkor?

Hol a hiba a kódrészletben?

```
main ()
{ struct else
  {
      int a;
      float b;
  };
struct else x;
}
function ()
{ struct else x;}
```

Mi történik a kódban?

```
int a = 1;
int main()
{
    int a = 2;
    int a = 3;
    printf("%d", a);
return 0;
}
```

Mi történik a kódban?

```
int main(void)
{
    char x = 'a';
    printf("%c\n", x);
    {
        printf("%c\n", x);
        char x = 'b';
        printf("%c\n", x);
    }
    printf("%c\n", x);
}
```

Hatáskörök

- Blokk hatáskör: { } között létrehozva
  - Beágyazott blokk: a blokkon belül létrehozott újabb blokkok: a bent deklaráltak kívül nem látszanak
- Függvény hatáskör: a függvényben létrehozott címke a függvényben mindenhol hivatkozható
- Program hatáskör (globális változók)
- Fájl hatáskör: a nagyobb méretű programok több fájlból állnak. A statikus globális változók a deklarálástól a fájl végéig látszódnak.

Tárolási osztályok

A tárolási osztályok alatt azokat a memória-területeket értjük, ahová adatokat tehetünk.

- Globális változók: data segment (fix mennyiségű adat)
- Lokális változók: stack segment (változó méretű, függvényhívásoknál)
- Dinamikusan kezelt: heap, vagy kupac (mutatókkal kezelt, malloc()-al foglalt)

A tárolási osztályt befolyásoló módosítók

- **auto**: blokk hatáskörű változóknál jelzi, hogy a változó ideiglenes, a hatáskör végén felszabadítható a területe. Alapértelmezett.
- **static**: blokknál és globális hatáskörnél jelzi a permanens tárolás igényét. Lásd: tömb visszaadása függvényből

A tárolási osztályt befolyásoló módosítók #2

- **register:** a gyorsabb feldolgozás érdekében kezdeményezzük a változó CPU regiszterbe kerülését.
- **extern:** más fájlokban létrehozott globális változók felhasználása a saját fájljainkban

Példa a static használatára

- Mit ír ki a program?

```
void kiir()
{
     static int i = 1;
     printf("A valtozo erteke: %d\n", i);
     i++;
}
int main(int argc, char *argv[])
{
    int j=0;
    while( j < 3)
    {
           kiir();
           j++;
    }
return 0;
}
```

A static hatása a függvényváltozókra

- A függvények static változói láthatóságban csupán lokálisak, élettartamukban viszont globálisak.

Parancssori argumentumok

- Az elindított programoknak is lehet paramétereket adni. Ezeket a main függvény mutatók tömbjében kapja meg. Ennek jelölése: `char *argv[]`, vagy `char **argv` (mutatók mutatója, jelentése: argument vector). Az első paraméter az `argc`, ami a szóközzel elválasztott paraméterek számát adja meg. Jelentése: argument count.

Parancssori paraméterek #2

- A paramétereket tartalmazó tömb 0. eleme a program neve lesz, az utolsó eleme pedig egy NULL érték. Emiatt:

  `argc[argv] = NULL`

Példa program írása

A program visszatérési értéke

- A main() függvény visszatérési értéke egy egész szám. Ezt a számot az operációs rendszer a program befejezésénél átadja annak a programnak, ami a miénket meghívta. A jelentése: hibakód, 0 = nincs hiba, 1, 2, … pedig valamilyen hiba.

Példa visszatérési értékre

```
#include <stdio.h>

int main(int argc, char *argv[]) {
int egyik, masik;

// 1-gyel több, mint a parameterek
if (argc-1 != 2) {
printf("%s: ket szamot adj meg!\n", argv[0]);
return 1; //nem ket parameter van: 1-es hibakod
}

if (sscanf(argv[1], "%d", &egyik) != 1) {
printf("Hibas elso parameter: %s!\n", argv[1]);
return 2; // 2-es hibakod: hibas parameter
}
```

Példa visszatérési értékre #2

```
if (sscanf(argv[2], "%d", &masik) != 1) {
printf("Hibas masodik parameter: %s!\n", argv[2]);
return 2;
}

printf("Az osszeguk: %d\n", egyik + masik);

return 0; // 0-s kod: minden rendben
}
```

Fájlok kezelése

- FILE * típusú pointerekkel kezelhetünk fájlokat (stdio.h szükséges

```
FILE *fp; // fájl mutató (filehandler)
fp = fopen("szoveg.txt", "wt"); // megnyitás módja

if (fp == NULL) {
     perror("szoveg.txt megnyitása"); //írunk az stderr-re
     return; // sajnos nem sikerült

fprintf(fp, "Hello, vilag!\n");
for (i = 0; i < 20; ++i)
fprintf(fp, "%d\n", i);

fclose(fp);
```

Fájlok megnyitási módjai

- Írás: w
- Olvasás: r
- Szöveges: t
- Bináris: b
- Hozzáfűzés: a
- Írás és olvasás: +

Az fopen() függvény visszatérési értéke olyan mutató, ami a fájlra hivatkozik. Az útvonal megadása esetében vagy /-t használjunk, vagy \\-t.

Hibás fájlnyitási kísérlet

- A perror(string) függvény visszaadja a szabványos hibakimenetre a legutóbb történt hiba okát. Ha a paraméterében adunk meg stringet, akkor azt is kiírja.

Bináris fájlok

- Fájlokból írásra és olvasásra használhatjuk az fread() és fwrite() függvényeket. Szintaxisuk hasonló:

fread(ptr, méret, db, fp);

fwrite(ptr, méret, db, fp);

- A ptr mutatja a memóriában lévő adatszerkezet helyét, a méret a méretét bájtban, a db az elemszámát, az fp pedig a fájlt, amiből/amibe olvasunk/írunk.
- A függvények visszatérési értéke a beolvasott/kiírt elemek száma

Összetett szerkezet bináris kiírása

```
struct adat {
char nev[13];
short eletkor;
} tomb[2];


strcpy(tomb[0].nev, „Adam");
tomb[0].eletkor = 20;
strcpy(tomb[1].nev, „Eva");
tomb[1].eletkor = 18;
```

Összetett szerkezet bináris kiírása #2

```
FILE *fp;

fp = fopen("adat.dat", "wb");
fwrite(tomb, sizeof(struct adat),
2, fp);
fclose(fp);
```

Hogyan néz a fájl tartalma annak létrejötte után?

Fájlmásoló program parancssori paraméterekkel

```
#include <stdio.h>

int main(int argc, char *argv[]) {
FILE *fbe, *fki; /* két fájl */
fbe = fopen(argv[1], "rb"); /* read binary */
fki = fopen(argv[2], "wb"); /* write binary */

char puf[1024]; /* puffer; char=bájt */
int olv; /* ahányat olvasott */
while ((olv = fread(puf, sizeof(char), 1024, fbe)) > 0)
fwrite(puf, sizeof(char), olv, fki);

fclose(fbe);
fclose(fki);

return 0;
}
```

Szöveges fájlok kezelése

- A bináris fájlok szerkezete architektúránként eltérhet: nem feltétlen egyforma méretű pl. az int minden platformon. Ennek elkerülésére használhatjuk az `fwrite()` függvényben a `sizeof(típus)` méretmegjelölést.
- A szöveges fájlokban eltérést okozhat a sor vége jelölése: a Linuxon egy bájt, Windows-on kettő. A kódolás továbbra is ASCII alapú: 1 karakter = 1 bájt.

Szöveges fájlok függvényei

- `fprintf(fájlpointer, formátumsztring, kifejezések)`

  A függvény visszatérési értéke a kiírt bájtok száma, hiba esetén negatív érték.

- `fscanf(fájlpointer, formátumsztring, kifejezések)`

  A függvény visszatérési értéke a beolvasott elemek száma, ez lehet kevesebb is, mint az elvárt, vagy akár nulla is.

Példa az fprintf()-re

```
FILE * fp;
fp = fopen ("file.txt", "w+");
fprintf(fp, "%s %s %s %d", "2", "x", "2=", 4);
fclose(fp);
```

Példa az fscanf()-re

```
char str1[10], str2[10], str3[10];
int eredmeny;
FILE * fp;
fp = fopen ("file.txt", "rt");
fscanf(fp, "%s %s %s %d", str1, str2, str3, &eredmeny);
printf("%s", str1 );
printf("%s", str2 );
printf("%s", str3 );
printf("%d", eredmeny );
fclose(fp);
```

Szöveges fájlok függvényei #2

- `fputs(kiírandó szöveg, fájlpointer)`
  A \0 kivételével kiírja a szöveget. A visszatérési érték siker esetén egy nem-negatív érték. Hiba esetében EOF, illetve hibajelzés.
- `fgets(célkaraktertömb kezdete,másolandó maximális karakterszám (\0-val), fájlpointer)`
  Beolvas szöveget, karakterszám-1-ig, vagy újsorig, esetleg fájl végéig. Siker esetében visszatér a szöveggel. Ha közben a fájl véget ért, EOF jelzi. Olvasási hiba esetében null pointerrel tér vissza és a hibajelző jelez.

Példa az fputs() használatára

```
FILE *fp;
fp = fopen("file.txt", "w+");
fputs("Első sor kiírása.", fp);
fputs("Második sor kiírása.", fp);
fclose(fp);
```

A függvényeknek van fputc, fgetc változata is, egyetlen karakter számára.

Példa az fgets()-re

```
FILE * pFile;
char mystring [100];
pFile = fopen ("file.txt" , "r");
if (pFile == NULL) perror ("Fajl nyitasi hiba");
else {
  if ( fgets (mystring , 100 , pFile) != NULL )
    puts (mystring);
  fclose (pFile);
}
```

Hibák észlelése

- `ferror(fájlmutató)`

Visszaadja a fájllal kapcsolatos hiba jelzés számát, különben nullát.

- `feof(fájlmutató)`

Nem-nulla értéket ad vissza, ha az ELŐZŐ, a fájllal kapcsolatos művelet során a fájl végére értünk, különben nullát ad.

- `clearerr(fájlmutató)`

Törli az előző indikátorok értékét.

Példa hibák jelzésére

```
FILE * pFile;
pFile = fopen("file.txt","r");
if (pFile==NULL) perror ("Hiba fajlnyitaskor");
else {
  fputc ('x',pFile);
  if (ferror (pFile)) {
    printf ("Hiba fajliraskor");
    clearerr (pFile);
  }
  fgetc (pFile);
  if (!ferror (pFile))
    printf ("Sikeres olvasas a fajlbol");
  fclose (pFile);
}
```

Mit csinál a program?

```
FILE *fp;
char s[100];
int i=1;
fp=fopen("hirek.txt","r");
if (fp!=NULL)
{
   while (fgets(s,100,fp)!=NULL)
   {
        if (i%3==0) puts(s);
        i++;
   }
   fclose(fp);
}
else { printf("Hiba történt a fájlnyitáskor!\n");}
```

Mik a hibák a programban?

```
FILE *fptr;
char nev[20];
fptr = fopen("emp.rec", "r");
if (fptr == NULL)
{
    printf("A fajl nem talalhato");
    return;
}
printf("Adja meg a nevet:\n");
scanf("%s", nev);
fprintf(fptr, "Nev= %s\n", nev);
```

Hibakezelés

- A nyelv nem tartalmaz közvetlen hibakezelést. Elsősorban a függvények visszatérési értékei jelzik a problémákat: -1, vagy null érték a legtöbb fügvvénynél. Ilyenkor egy `errno` globális változó jelzi a hibakódot. Az `errno.h` számos hibakódot tartalmaz. Jó gyakorlat a program indításakor lenullázni az errno-t, majd szükség esetén újra leellenőrizni az értékét.

Hibakezelés #2

- `perror()` függvény

  a saját üzenetünk + a jelenlegi errno szöveges leírása
- `strerror()` függvény

  a jelenlegi errno szöveges leírására mutat
- `stderr`

  szabványos hibakimenet, ne felejtsünk el ott is jelezni!

Próbáljuk ki!

```
FILE * pf;
int errnum;
pf = fopen ("nemletezo.txt", "rb");
if (pf == NULL) {
  errnum = errno;
  fprintf(stderr, "Hibakod: %d\n", errno);
  perror("Perror hibauzenete");
  fprintf(stderr, "Fajlnyitasi hiba: %s\n", strerror(
   errnum ));
 }
 else {
  fclose (pf);
 }
```

Program kilépési kódja makróval

- `EXIT_SUCCESS`
  sikeres programfutás eredménye, az értéke 0

- `EXIT_FAILURE`
  hibás futtatás eredménye, az értéke -1

Mi az alapértelmezett stderr?

```
main() {
    int osztando = 20;
    int oszto = 5;
    int hanyados;
    if( oszto == 0) {
        fprintf(stderr, "Nullaval osztas!\n");
        exit(EXIT_FAILURE);
    }
    hanyados = osztando / oszto;
    fprintf(stderr, "Hanyados : %d\n", hanyados );
    exit(EXIT_SUCCESS);
}
```

Példák hibakódokra

```
#define ENOENT       2  /* No such file or directory */
#define ENOMEM      12  /* Out of memory */
#define EACCES      13  /* Permission denied */
#define EBUSY       16  /* Device or resource busy */
#define ENOTDIR     20  /* Not a directory */
#define ENOSPC      28  /* No space left on device */
#define EROFS       30  /* Read-only file system */
```

Mit csinál a program? Van-e hibája?

```
FILE *fp;
int a[10], i, x=0;
fp=fopen("adatok.dat","rb");
if (fp!=NULL)
{
  fread(a,sizeof(int),10,fp);
  for (i=0;i<15;i++) x*=a[i];
  printf("Eredmeny: %d",x);
  fclose(fp);
}
else printf("Hiba fajlnyitaskor!\n");
```

Pointerek, függvényparaméterezés

- Ismételjük át a függvények esetében a cím szerinti paraméterátadás működését!

```
void noveld(int *mit)
{
   *mit += 1;
}
```

Pointer, mint tömb

- A pointert nemcsak az eddig megtanult módon használhatjuk, de tömbszerűen is hivatkozhatunk rá.

```
int tomb[4], *pi;
pi = tomb; /* &tomb[0] */
pi[2] = 4; /* *(pi+2)=4 */
tomb[1] = 2;
```

Tömbátadás függvényeknek

- Akár változót adunk át cím szerint, akár egy tömböt a függvénynek, mindkét esetben egy pointer adódik át.

```
void tomb_kiir(int *tomb, int meret) {
    int i;
    for (i = 0; i < meret; ++i)
         printf("%d ", tomb[i]);
    }

int tomb[10];
tomb_kiir(tomb, 10);
```

Egyenértékű paraméterátadások

- Teljesen egyforma a fordító számára az alábbi kétféle paraméterátadás:

- `void fuggveny(int*);`
- `void fuggveny(int[]);`

- Vagyis a tömb nem másolódik át, csak a címét kapjuk meg (hatékonyság miatt)!

A const kulcsszó szerepe

- Akadályozzuk meg, hogy a függvény (akár véletlenül is) megváltoztassa a kapott tömb elemeit!
- `int const *` – ez azt jelenti, hogy ez a mutató konstans integerekre mutat. Vagyis hogy lehet hivatkozni a mutatott értékeket (a tömb elemeit), és ki is lehet olvasni azokat, de változtatni nem lehet rajtuk.

```
void kiir(int const *tomb, int meret);
int tomb[100] = {.., .., .., ..};
kiir(tomb, 100);
```

A const további haszna

- Több helyen hivatkozott, fix értékeket elég egy helyen megadni velük, utána nem kell több helyen átírni őket:

```
int const Magas = 480;
int const Szeles = Magas/3*4;
Szin const zold = { 133, 224, 89 };
```

A dinamikus memóriakezelés szükségessége

- Tömbök problémája: fix méretet kell előre meghatározni. Ha nem tudjuk előre, akkor saccolás:
  - Túl kevés: másolni kell
  - Túl sok: felesleges erőforrások, vagy el sem fér a veremben
- Változók élettartama nem szabályozható finoman
  - Globális változók: folyamatosan léteznek
  - Lokális változók: csak az adott függvényben

Dinamikus memóriakezelés

- A „*dinamikus memóriakezelés*” azt jelenti, hogy egyes memóriaterületek foglalását és felszabadítását mi vezéreljük a programból. A „dinamikus” szó az időre utal.
- Cserébe a mi felelősségünk nemcsak a foglalás, de a felszabadítás is! Ha elfelejtjük, memóriapazarló lesz a programunk és az operációs rendszer is leállíthatja.

A malloc() függvény

- `void *malloc(size_t méret)`

  Lefoglal egy bájtban megadott méretű memóriaterületet (malloc: memory allocation).

  - Visszaad egy pointert a legfoglalt területre, ami méret bájtból áll.
  - A terület inicializálatlan (memóriaszemét).
  - Azért void*, mert nem ismeri a típusunkat.
  - Ha nem sikerül, akkor NULL pointert ad.

A free() függvény

- `void free(void *ptr)`

  Felszabadít egy memóriaterületet, amit a malloc() foglalt. A *malloc() által adott* címet kell neki adni.

Fontos szabályok a foglaláshoz

- A lefoglalt memóriaterületet fel kell szabadítani. Ahány `malloc()` hívás, annyi `free()` hívás kell történjen.
- A `malloc()` által adott pointerre vigyázni kell! Ha elveszítjük, nem tudjuk majd azt a területet visszaadni.
- Allokálatlan memóriaterület nem használható!
- Nem szabad dinamikus tömbön a `sizeof` operátort használni – az nem a tömb méretét, hanem a pointer méretét fogja megadni!

Mi a hiba a példákban?

- ```
  char *szoveg;
  szoveg[2] = 'x';
  ```

- ```
  free(szoveg);
  szoveg[2] = 'x';
  ```

Type cast

- Egy adott kifejezés elé zárójelbe írt típus, ezzel jelezzük, hogy az adott kifejezést miként szeretnénk használni

```
double *tomb;
tomb = (double*) malloc(100*sizeof(double));
```

- A malloc void* mutatót ad vissza, mivel nem tudja, milyet szeretnénk használni. Ezt alakítjuk át az esetünkben double* típusúra.

Egy jó példa dinamikus foglalásra

```
#include <stdlib.h>

double *tomb; /* ptr a majdan lefoglalt területre */
int i, n;

printf("Hany szamot? ");
scanf("%d", &n);

tomb = (double*) malloc(n*sizeof(double)); /* foglalás */
if (tomb == NULL) {
printf("Nem sikerult memoriat foglalni!\n");
return 1;
}

tomb[3] = 12; /* ez egy tömb, már használhatjuk!  */

free(tomb); /* felszabadítás */
tomb = NULL; /* kell ez? */
```

További foglalási szabályok

- A `malloc()`-hoz a méretet nekünk kell kiszámítani. A lefoglalt terület memóriaszemetet tartalmaz.
- A pointereket az indexelő operátorral használhatjuk tömbként, a függvényeket is.
- A `free()` függvény elvileg egy void* típust vár, de az adott típusról erre automatikusan konvertálunk.
- Felszabadítás után már tilos hivatkozni rá!

Az élettartam korlátai

- Miért hibás a következő kód?

```
int *fv(void) {
int i = 7;
return &i; /* hibás!!! */
}

int *ptr = fv();
```

Az élettartam dinamikus szabályozása

```
/* visszatér egy sztringgel, ami s1 és s2 összefűzve.
* a hívónak fel kell szabadítania a kapott sztringet! */
char *osszefuz(char const *s1, char const *s2) {
int mennyi;
char *res;

mennyi = strlen(s1)+strlen(s2)+1;
res = (char*) malloc(mennyi * sizeof(char));
if (res == NULL) /* ha nem sikerült a malloc() */
return NULL;

strcpy(res, s1);
strcat(res, s2);

return res;
}
```

Függvények dinamikus eredményei

- A lokális `char* res` változó ugyan megszűnik, de azt a visszaadáskor lemásoljuk! (Csak a pointert! Nem az egész tömböt!)
- Amikor visszatérünk, a létrehozott tömb így *nem szűnik meg!* A pointer ugyan igen, de azt *visszaadjuk*,és a hívónak azt el kell tárolnia magának.

```
char *s;
s = osszefuz("alma", "fa");
printf("Ezt írtad be: %s\n", s);
free(s); /* ! */
s = NULL; /* van, aki direkt kinullázza utána */
```

A calloc() függvény

- `void *calloc(size_t darabszám, size_t egy_elem)`

  Ha nemcsak foglalni szeretnénk, de kinullázni is a bájtokat. Általában csak egész típusokhoz jó ezért.

```
int *tomb = (int*) calloc(100, sizeof(int));
```

A realloc() függvény

- void *realloc(void *ptr, size_t méret)
  Újrafoglalja, átméretezi a ptr-rel megadott dinamikus tömböt. A meglévő adatokat másolja. Ha nagyobb lett, a többi memóriaszemét, ha kisebb, a vége elveszik. Ha nem fér el az adott memóriaterületen, akkor másolva lesz, ami lassú lehet.

Példa: dinamikus tömb (részlet)

```
struct DinTomb {
  double *adat;
  int meret;
};
DinTomb dt; DinTomb *pdt;
pdt->meret = meret;
pdt->adat = (double*)
  malloc(meret*sizeof(double));
```

A nyíl operátor

- A nyíl operátor a pointer által mutatott struktúra adattagja. Egyenértékű utasítások:

```
pdt->meret = 160;(*pdt).meret = 160;
```

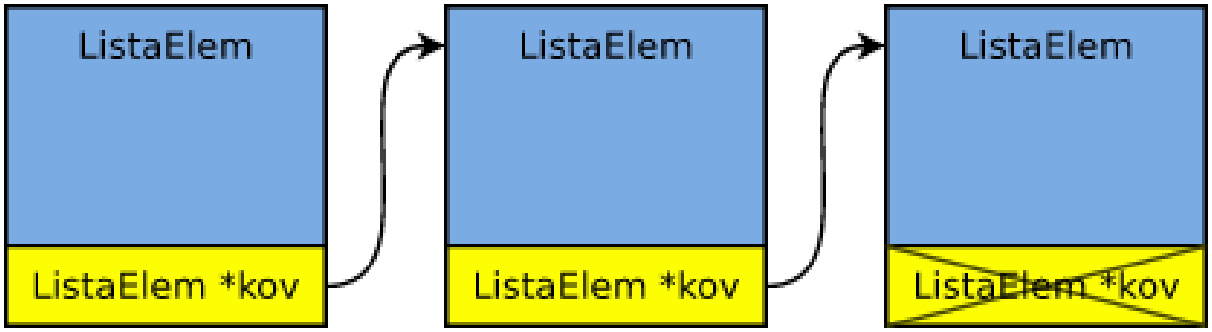
- A zárójel a . operátor magasabb precedenciája miatt kell. A nyíl operátor ezt feleslegessé teszi.

Dinamikus tömbök problémái

- A tömb használatának előnye:
  - Gyors, könnyen kiszámítható adatelérés (pl. logaritmikus keresésnél)
- A tömb hátrányai:
  - Összefüggő adatterületet igényel, a szabad, rendelkezésünkre álló memóriaterület nem feltétlenül összefüggő
  - Nehézkes méretnövelés: újra kell foglalni, majd másolni, tehát egy ideig kétszer is szerepel ugyanaz az adat a tárban.

Láncolt lista

- Adatszerkezet, ahol az egyes elemek láncba vannak fűzve azáltal, hogy tárolják a soron következő elem címét.



Láncolt lista megvalósítása

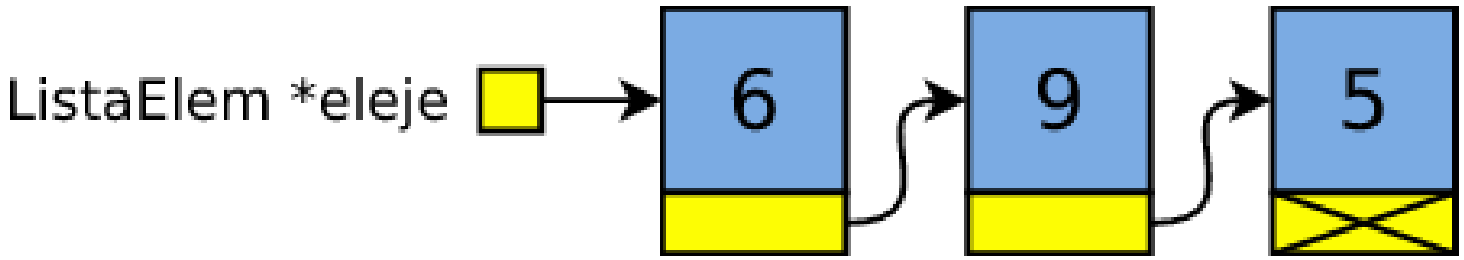
```
typedef struct ListaElem {

  … /* tetszőleges adat(ok) */
  struct ListaElem *kov;

} ListaElem;
```

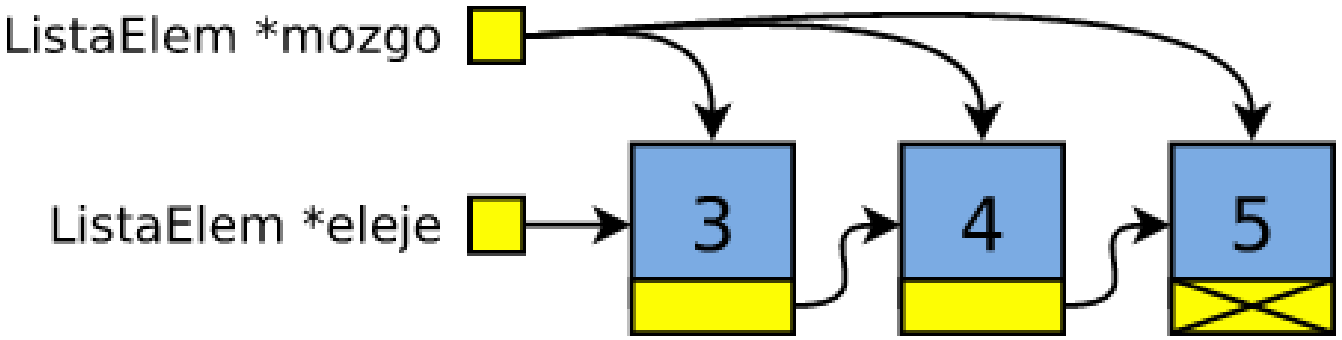
A lista nyilvántartása

- A bejáráshoz szükséges az első elem címének eltárolása.
- A lista végét az utolsó elem mutatójában lévő NULL érték jelzi.

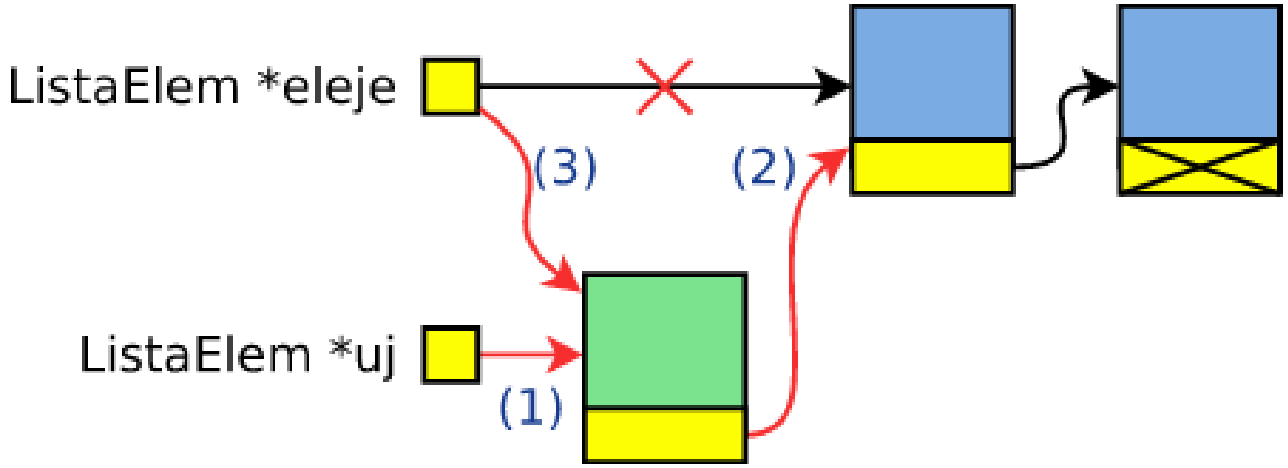


A lista bejárása

```
ListaElem *mozgo;
for (mozgo=eleje; mozgo!=NULL; mozgo=mozgo->kov)
{
  printf("%d", mozgo->szam);
}
```



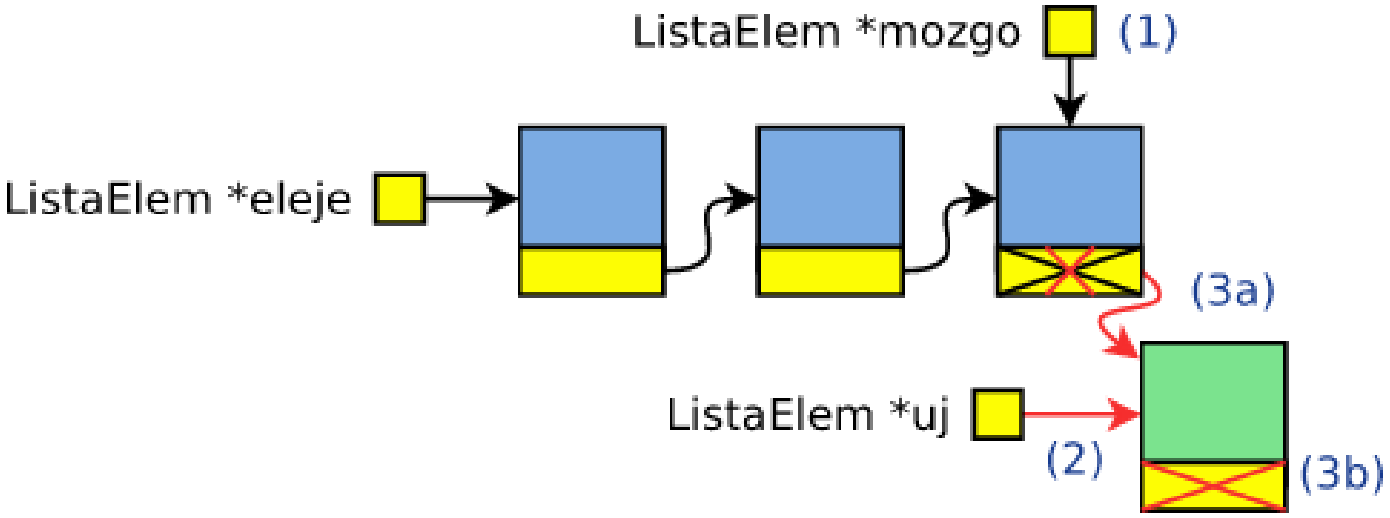
Listaműveletek: beszúrás az elejére




```
ListaElem *uj;

uj = (ListaElem*) malloc(sizeof(ListaElem)); /* 1 */
uj->kov = eleje; /* 2 */
eleje = uj; /* 3 */
```

Hozzáfűzés a végéhez




```
ListaElem *mozgo, *uj;

for (mozgo = eleje; mozgo->kov != NULL; mozgo = mozgo->kov)
/* 1 */
; /* üres ciklus */
uj = (ListaElem*) malloc(sizeof(ListaElem)); /* 2 */
mozgo->kov = uj;
uj->kov = NULL;
```

Lista felszabadítása

- Miért hibás a következő próbálkozás?

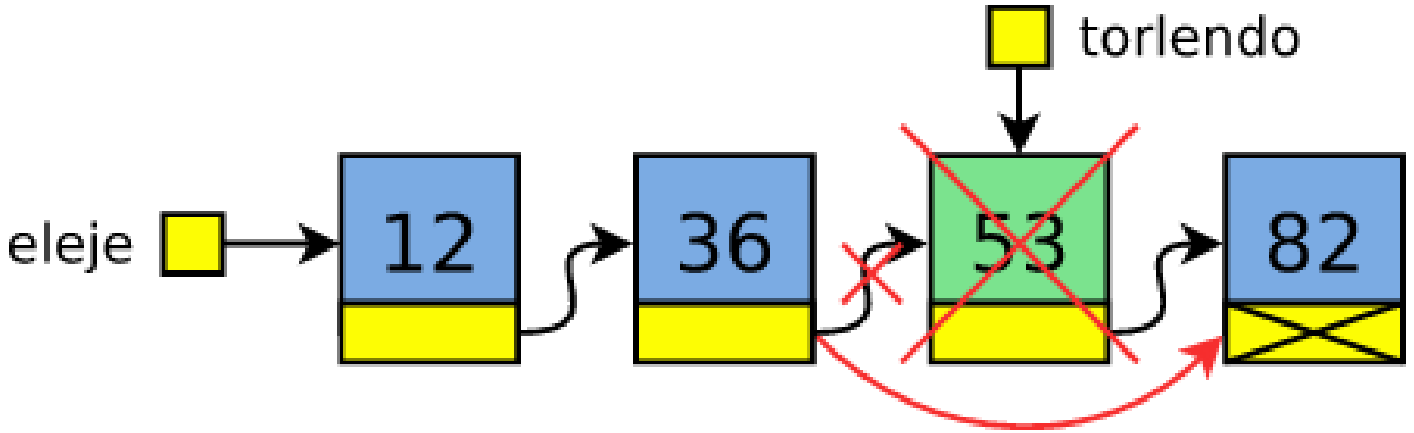
```
for (iter = eleje; iter != NULL;
  iter = iter->kov)
{
  free(iter);
}
```

A lista helyes felszabadítása

```
iter = eleje;
while (iter != NULL) {
   ListaElem *temp = iter->kov; /* következő elem */
   free(iter);
   iter = temp;
}
eleje = NULL;
```

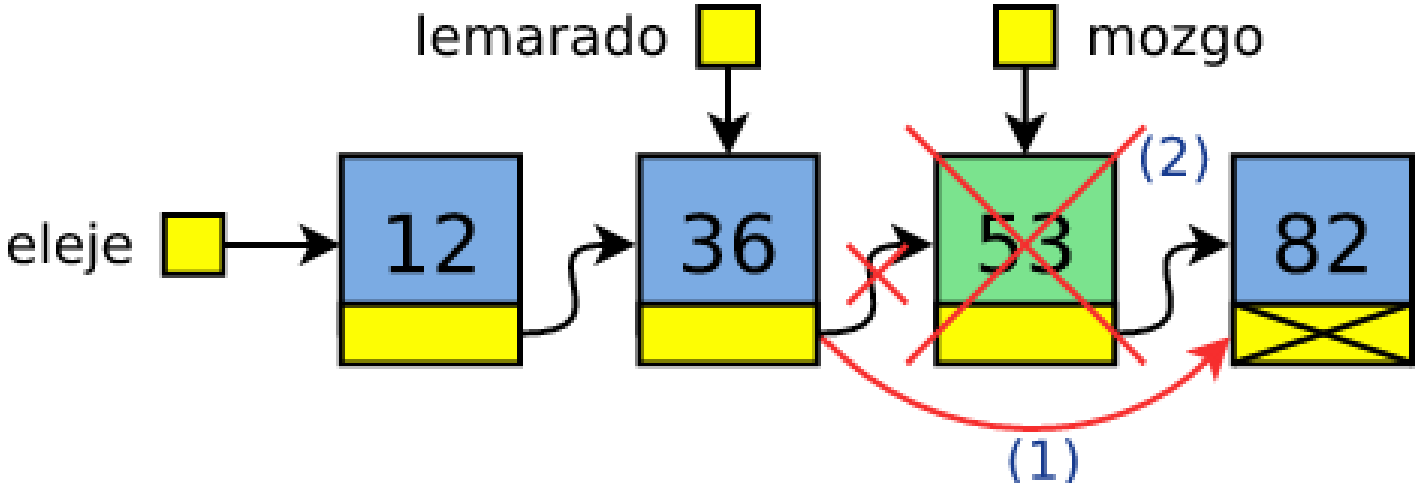
Elem törlése a listából

- Ha a lista első elemét töröljük, akkor módosítani kell az „eleje” nevű pointert.
- Törlés lista belsejéből:



Probléma, hogy a törlendő elem előtti elem mutatóját kellene módosítani, visszafelé haladni pedig nem lehet.

Elem törlése a listából #2




```
lemarado = NULL; mozgo = eleje;
while (mozgo != NULL && mozgo->adat != keresett) {
lemarado = mozgo; mozgo = mozgo->kov;
}
lemarado->kov = mozgo->kov; /* törlendő = ahol megállt */
free(mozgo);
```