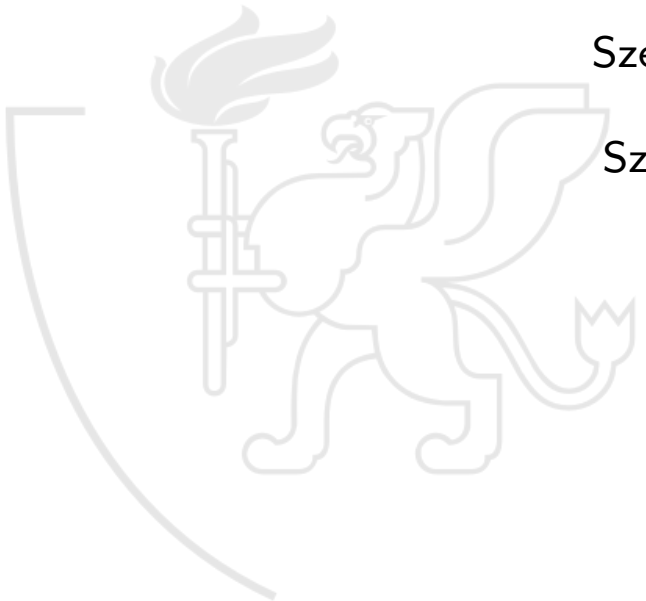
Programozás Alapjai

Dr. Gergely Tamás
Dr. Jász Judit

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék



2023

(v1016)

Tartalom

Tartalom

Az algoritmus vezérlése

Algoritmus vezérlése — *Az az előírás, amely az algoritmus minden lépésére (részműveletére) kijelöli, hogy a lépés végrehajtása után melyik lépés végrehajtásával folytatódjék (esetleg fejeződjék be) az algoritmus végrehajtása.*

- Az algoritmusnak, mint műveletnek a vezérlés a legfontosabb komponense.
- Ezt az előírást nevezzük az algoritmus vezérlésének.

Vezérlési módok

- A vezérlési mód azt fejezi ki, hogy egyszerűbb műveletekből hogyan építünk fel összetett műveletet és ennek milyen lesz a vezérlése.
- Négy fő vezérlési módot különböztetünk meg:

Szekvenciális Véges sok adott művelet rögzített sorrendben egymás után történő végrehajtása.

Szelekciós Véges sok rögzített művelet közül adott feltétel alapján valamelyik végrehajtása.

Ismétléses Adott művelet adott feltétel szerinti ismételt végrehajtása.

Eljárás Adott művelet alkalmazása adott argumentumokra, ami az argumentumok értékének pontosan meghatározott változását eredményezi.

Algoritmusok leírása

- Az algoritmusok leírására többféle módszer használható
  - Természetes nyelvi leírás
    - Nagyon távol állhat a „géptől”.
  - Pszeudokód
    - „Majdnem” programozási nyelv, struktúrált, de sokkal szabadabb mint egy valódi programozási nyelv.
  - Folyamatábra
    - Grafikus, kevésbé struktúrált (tetszőleges vezérlésátadás), a működési folyamatra koncentrál.
  - Szerkezeti ábra
    - Grafikus, struktúrált, az algoritmus felépítését és a működési folyamatot is leírja.

Folyamatábra

- Ha csak a kész algoritmus működését akarjuk leírni, és a szerkezete kevésbé fontos, akkor használhatjuk a *folyamatábrát*, mint leíró nyelvezetet.
  - Ez olyan ábra, amelyben az algoritmus egyes lépéseit és a lépések közötti vezérlési viszonyt szemléltethetjük irányított vonalakkal.
  - Az a mód, ahogyan a folyamatábra leírja az algoritmust nagyon közel áll az alacsony szintű programozáshoz (assembly nyelv).

Folyamatábra

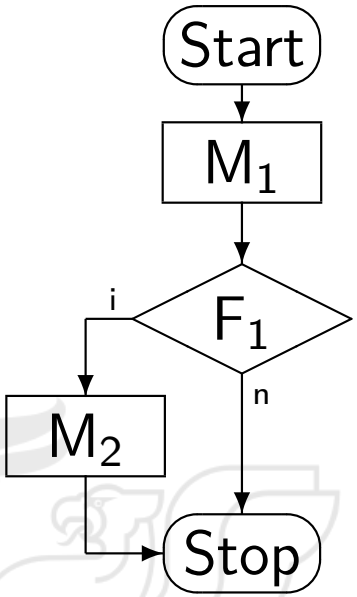
Szintaxis

- Legyenek $M = \{M_1, \ldots, M_k\}$ műveletek és $F = \{F_1, \ldots, F_l\}$ feltételek. Az (M, F) feletti folyamatábrán olyan irányított gráfot értünk, amelyre teljesül a következő 5 feltétel:
  1. Egy olyan pontja van, amely a *Start* üres művelettel van címkézve és ebbe a pontba nem vezet él.
  2. Egy olyan pontja van, amely a *Stop* üres művelettel van címkézve és ebből a pontból nem indul él.
  3. Minden pontja M-beli művelettel vagy F-beli feltétellel van címkézve, kivéve a *Start* és a *Stop* pontokat.
  4. Ha egy pont
     - M-beli művelettel van címkézve, akkor belőle egy él indul ki,
     - ha F-beli feltétellel van címkézve, akkor belőle két él indul ki és ezek az i (igen) illetve n (nem) címkét viselik.
  5. A gráf miden pontja elérhető a *Start* címkéjű pontból.

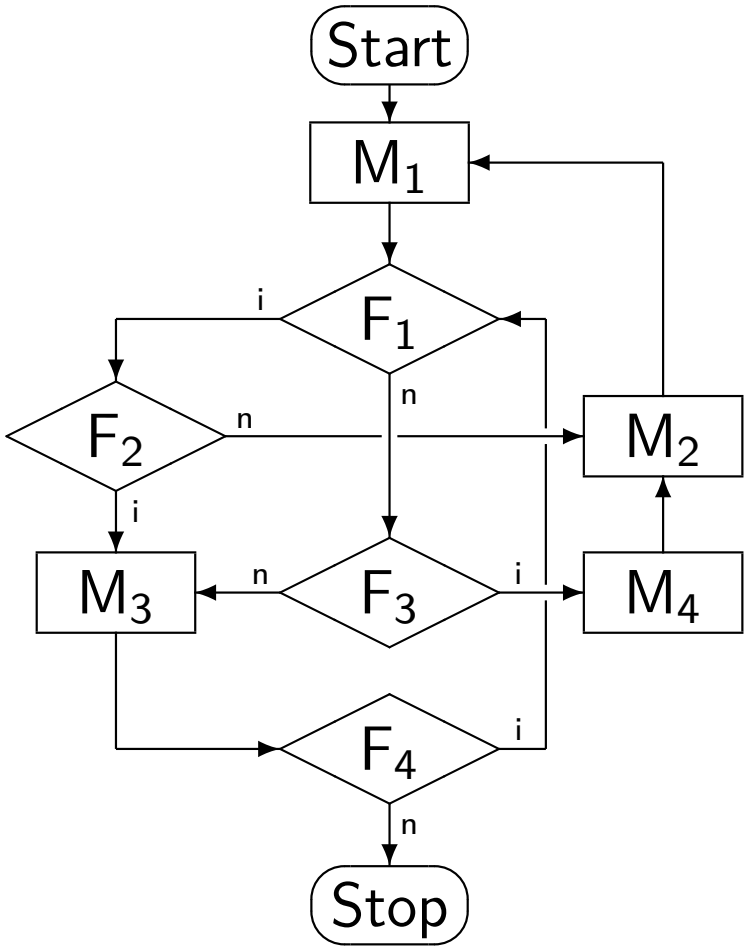
Folyamatábra

Példa

- Egy egyszerű folyamatábra:



- És egy bonyolultabb:



Folyamatábra

Szemantika

- Egy folyamatábra a következő összetett vezérlési előírást jelenti:
  - A végrehajtás a *Start* pontból indul.
  - Az összetett művelet végrehajtása akkor ér véget, ha a *Stop* pont kap vezérlést.
  - A gráf egy pontjának végrehajtása azt jelenti, hogy
    - Ha a pontban M-beli művelet van, akkor a művelet végrehajtódik és a vezérlés a gráf azon pontjára kerül, amelybe a pontból kiinduló él vezet.
    - Ha a pont F-beli feltétellel van címkézve, akkor kiértékelődik a feltétel. Ha értéke igaz, akkor az a pont kap vezérlést, amelybe az *i* (igen) címkéjű él vezet, egyébként az a pont kap vezérlést, amelybe az *n* (nem) címkéjű él vezet.

Szerkezeti ábra

- Az algoritmus tervezése során *szerkezeti ábrával* (*structure diagram*, *struktúradiagram*) (is) le tudjuk írni, hogy a problémát milyen részproblémákra bontottuk, és a megoldásukat milyen módon raktuk össze.
- A szerkezeti ábra egyszerre fejezi ki az algoritmustervezés folyamatát és a kifejlesztett algoritmust is.
- Egy részprobléma megoldását leíró szerkezeti ábrarész különálló ábrával is kifejezhető, amelynek gyökerében a részprobléma megnevezése áll.
- Minden vezérlési módhoz bevezetünk egy szerkezeti ábra jelölést. Az egyes vezérlések szerkezeti ábra jelöléseit a vezérléseknél mutatjuk be.

Tartalom

Eltelt idő kiszámítása

Problémafelvetés és specifikáció

- Problémafelvetés
  - Egy munkahelyen a dolgozóknak minden be- és kilépési időpontot regisztrálni kell. A munkáltató tudni szeretné, ki mennyi időt töltött bent, illetve ebből mennyi volt az ebédszünet. Ehhez kellene neki egy program, amivel a nap két időpontja között eltelt időt tudja kiszámolni.
- Specifikáció
  - Perc pontossággal dolgozunk.
  - A bemenő adat két időpont óra és perc formában, jelöljük ezeket O_1, P_1 illetve O_2, P_2-vel.
  - A bemeneti feltétel:
    - $0 \leq O_1 < 24$ és $0 \leq P_1 < 60$
    - $0 \leq O_2 < 24$ és $0 \leq P_2 < 60$
    - $O_1 < O_2$ vagy ($O_1 = O_2$ és $P_1 \leq P_2$)
  - A kimenő adat a két bemenő időpont között eltelt idő óra, perc formában, jelöljük ezeket O és P-vel.
  - A kimeneti feltétel:
    - $0 \leq O < 24$ és $0 \leq P < 60$

Eltelt idő kiszámítása

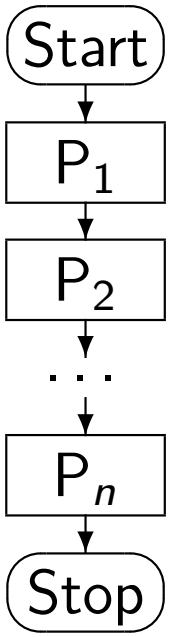
Algoritmustervezés

- Számolhatunk kétféleképpen:
  - külön kiszámítjuk az eltelt órákat és perceket, de akkor a végén esetleg korrigálni kell a számítást a negatív percek miatt, vagy
  - eleve percben számolunk, és csak a kiíratáskor alakítjuk át az eredményt óra:perc formátumra.
- Az eltelt idő percben kifejezve $(60 \cdot O_2 + P_2) - (60 \cdot O_1 + P_1)$.
- Tehát O, P akkor és csak akkor megoldás, ha
  - $60 \cdot O + P = (60 \cdot O_2 + P_2) - (60 \cdot O_1 + P_1)$, és
  - $0 \leq P < 60$
- Ez tulajdonképpen a kimeneti feltételek formális megadása, de az első feltétel már mindenképpen az algoritmustervezés fázisához kapcsolódik.

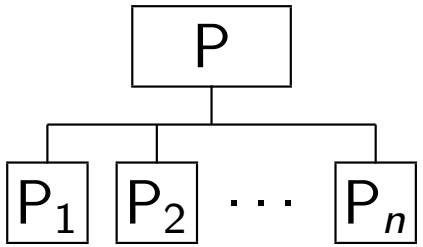
Szekvenciális vezérlés

- Szekvenciális vezérlésről akkor beszélünk, amikor a P probléma megoldását úgy kapjuk, hogy a problémát $P_1, \ldots, P_n$ részproblémákra bontjuk, majd az ezekre adott megoldásokat (részalgoritmusokat) sorban egymás után hajtjuk végre.
  - $P_1, \ldots, P_n$ lehetnek elemi műveletek, vagy nem elemi részproblémák megnevezései. Utóbbi esetben a részproblémát tovább kell bontani.

- Folyamatábrája:



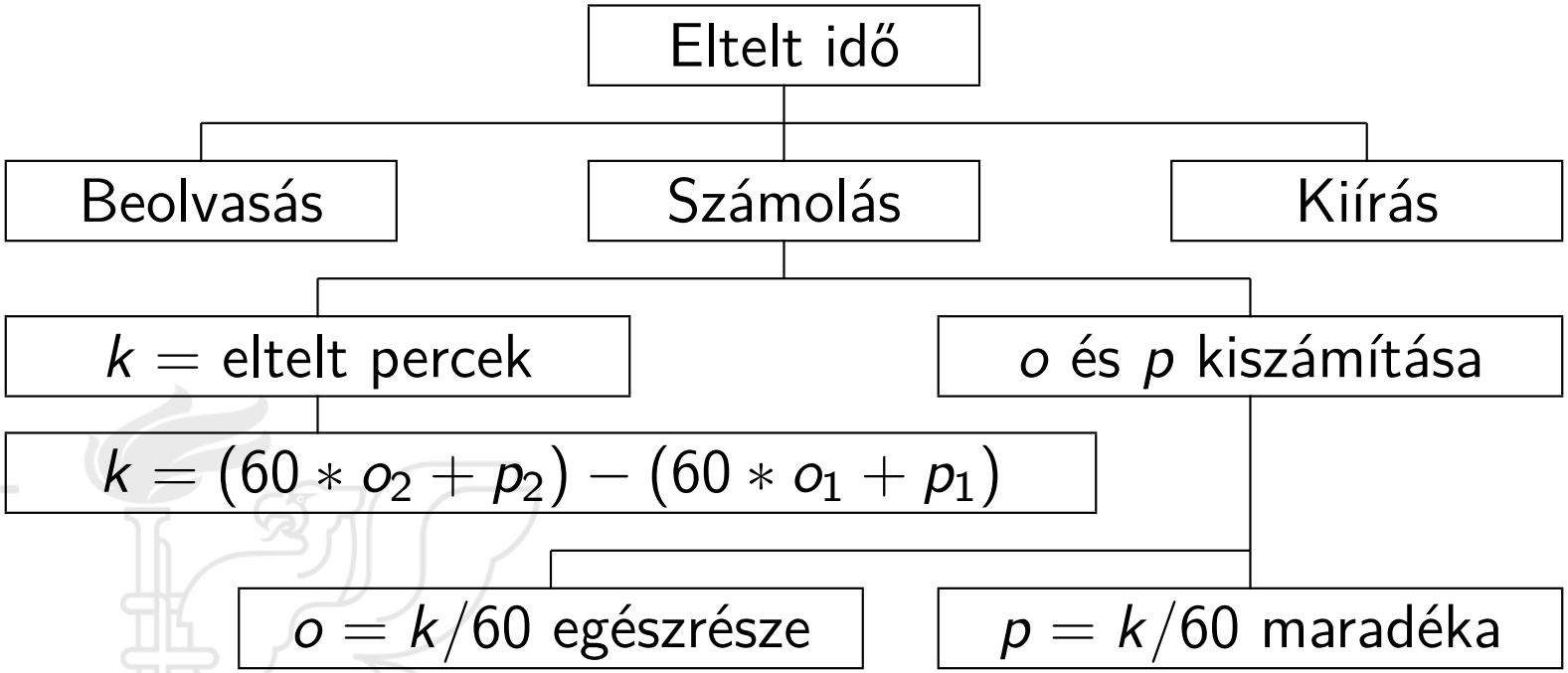
- Szerkezeti ábrája:



Eltelt idő kiszámítása

Algoritmustervezés – Szerkezeti ábra

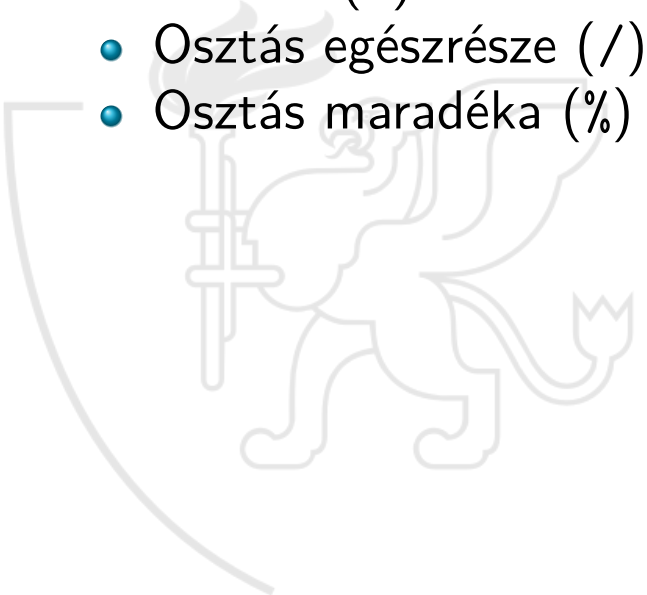
- Az eltelt idő probléma megoldásának szerkezeti ábrája:



Eltelt idő kiszámítása

Algoritmustervezés – Változók

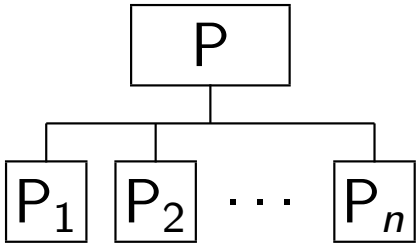
- Az Eltelt idő algoritmusban használt változók mindegyike egész típusú, értékük tetszőleges egész szám lehet
  - O_1, P_1, O_2, P_2, O, P, K
- Az egész értékeken a következő műveleteket alkalmaztuk
  - Összeadás (+)
  - Kivonás (−)
  - Szorzás (∗)
  - Osztás egészrésze (/)
  - Osztás maradéka (%)



Szekvenciális vezérlés

Megvalósítás C nyelven

- Szerkezeti ábrája:



- Megvalósítása:

```
{
    P1;
    ...
    Pn;
}
```

Eltelt idő kiszámítása [1/1]

eltelt.c [1–24]

```c
/* Egy nap két időpontja között mennyi idő telt el.
 * 1997. Szeptember 26.  Dévényi Károly, devenyi@inf.u-szeged.hu
 */

#include <stdio.h>

int main() {
    int o1, p1;                          /* az első időpont */
    int o2, p2;                         /* a második időpont */
    int o, p;                              /* az eltelt idő */
    int k;                           /* az eltelt idő percben */
    /* beolvasás */
    printf("Kérem az első időpontot óra perc formában\n");
    scanf("%d %d", &o1, &p1);
    printf("Kérem a második időpontot óra perc formában\n");
    scanf("%d %d", &o2, &p2);
    /*  számítás */
    k = 60 * o2 + p2 - (60 * o1 + p1);
    o = k / 60;
    p = k % 60;
    /*  kiíratás */
    printf("Az eltelt idő: %d óra %d perc.\n", o, p);
    return 0;
}
```

Tartalom

Eljárásvezérlés

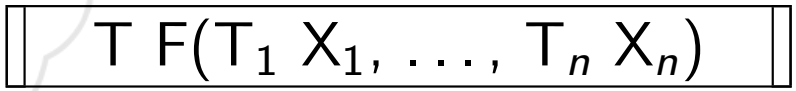
- Eljárásvezérlésről akkor beszélünk, amikor egy műveletet adott argumentumokra alkalmazunk, aminek hatására az argumentumok értékei pontosan meghatározott módon változnak meg.
- Az eljárásvezérlés fajtái:
  - Eljárásművelet
  - Függvényművelet
- Vannak olyan programozási nyelvek, ahol a függvény és eljárásműveletek jelentősen meg vannak különböztetve. Mivel azonban e kurzus keretében csak a C nyelvről lesz szó – ahol a különbség minimális –, egyelőre csak a függvényművelettel foglalkozunk, és a szerkezeti ábrán is igazodni fogunk a C nyelvhez.

Függvényművelet

- A matematikai függvény fogalmának általánosítása.
- Ha egy részprobléma célja egy érték kiszámítása adott értékek függvényében, akkor a megoldást megfogalmazhatjuk függvényművelettel.
- A függvényművelet specifikációja tartalmazza:
  - A művelet elnevezését
  - A paraméterek felsorolását
  - Mindegyik paraméter adattípusát
  - A művelet hatásának leírását
  - A függvényművelet eredménytípusát

Függvényművelet

- A függvényművelet jelölésére a `T F(T1 X1, ..., Tn Xn)` formát használjuk, ahol
  - `T` a függvényművelet eredménytípusa
  - `F` a függvényművelet neve
  - $\texttt{T}_i$ az i. paraméter adattípusa
  - $\texttt{X}_i$ az i. paraméter azonosítója
- A zárójeleket üres paraméterlista esetén is ki kell tenni.
- A C jelölésmódhoz igazodva, a függvény algoritmusa egy olyan szerkezeti ábrával adható meg, melynek a feje így néz ki:



- Továbbá a szerkezeti ábrában lennie kell (legalább) egy olyan `return` utasításnak, amely visszaadja a függvény által kiszámított értéket.

Függvényművelet

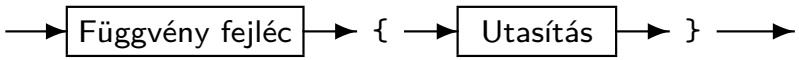
- A fent jelölt függvényműveletnek adott $A_1, \ldots, A_n$ aktuális argumentumokra történő végrehajtását függvényhívásnak nevezzük és az `F(`A_1`, ..., `A_n`)` jelölést használjuk.
- A függvényhívás kifejezés.
- A zárójeleket paraméter nélküli függvény hívása esetén is ki kell tenni.

Függvény

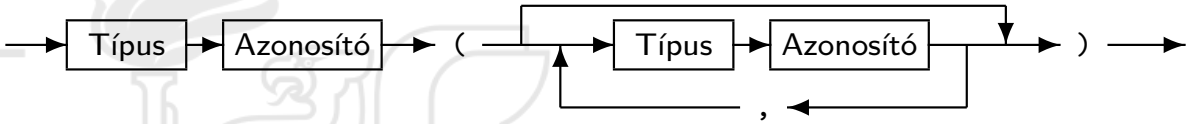
Szintaxis

- Függvény deklaráció

- Függvény definíció (egyben deklaráció is)



- Függvény fejléc



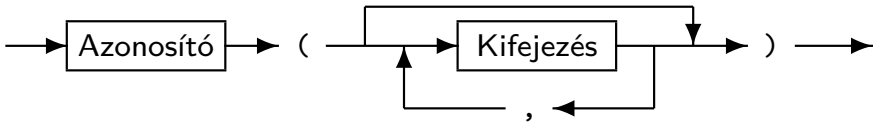
A `return` utasítás

- Minden függvényben szerepelnie kell legalább egy `return` utasításnak.
- Ha a függvényben egy ilyen utasítást hajtunk végre, akkor a függvény értékének kiszámítása befejeződik. A hívás helyén a függvény a `return` által kiszámított értéket veszi fel.
- A `return` utasítás szintaxisa C-ben

Függvényhívás

Szintaxis

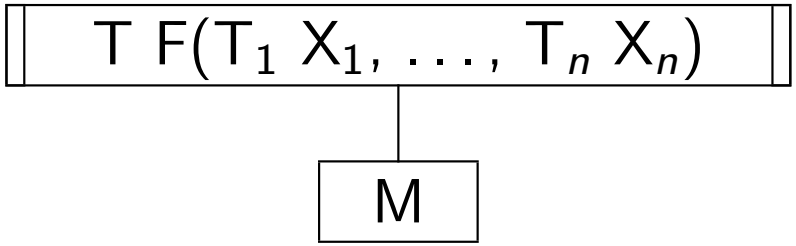
- Függvényhívás szintaxisa C-ben



Függvényművelet

Megvalósítás C nyelven

- Szerkezeti ábrája:



- Megvalósítása:

```
T F(T1 X1, ..., Tn Xn)
{
    M;
}
```

Függvényművelet

Példa

- A szabvány szerinti deklaráció:

```
double atlag(double a, double b) {
    return (a + b) / 2;
}
```

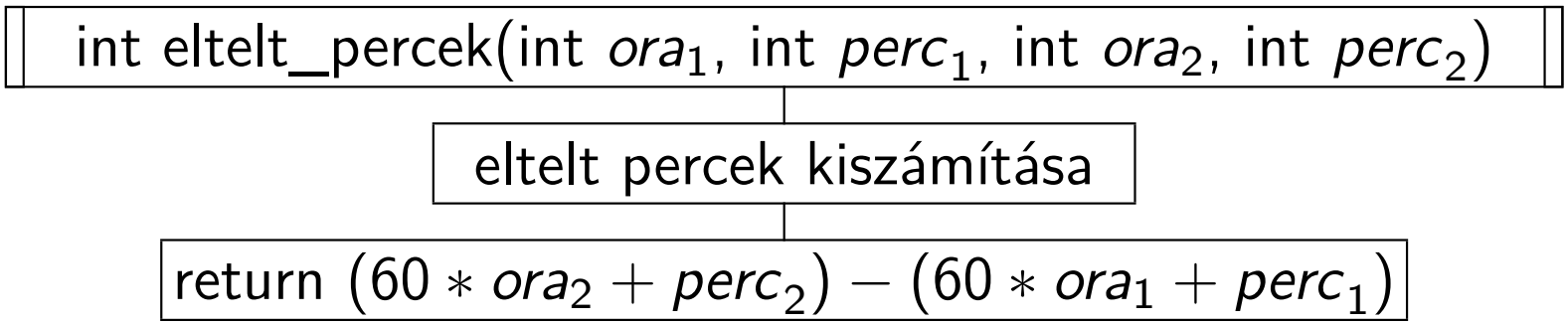
- Régen C-ben így kellett függvényt deklarálni:

```
double atlag(a,b)
    double a, double b;
{
    return (a + b) / 2;
}
```

Eltelt idő kiszámítása

Szerkezeti ábra 1.

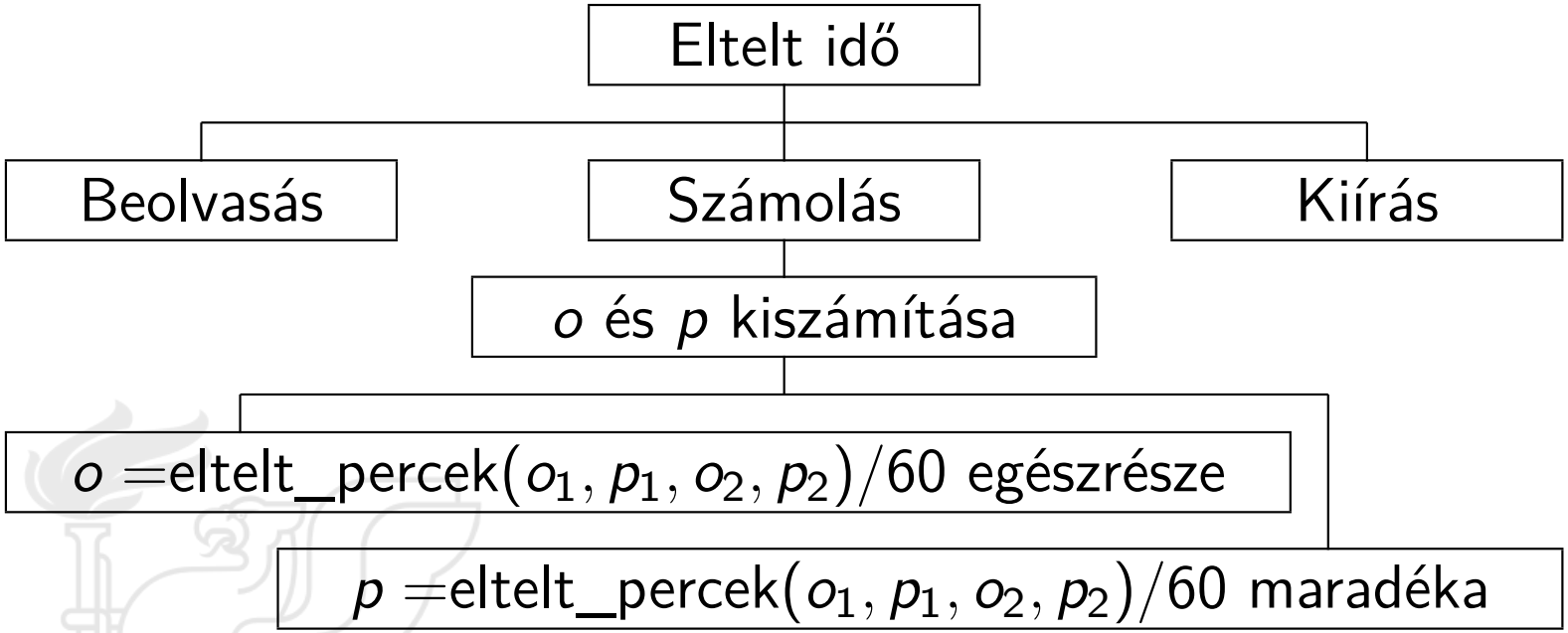
- Az eltelt perceket számító függvény szerkezeti ábrája:



Eltelt idő kiszámítása

Szerkezeti ábra 2.

- Az eltelt idő programjának szerkezeti ábrája:



Eltelt idő kiszámítása [1/1]

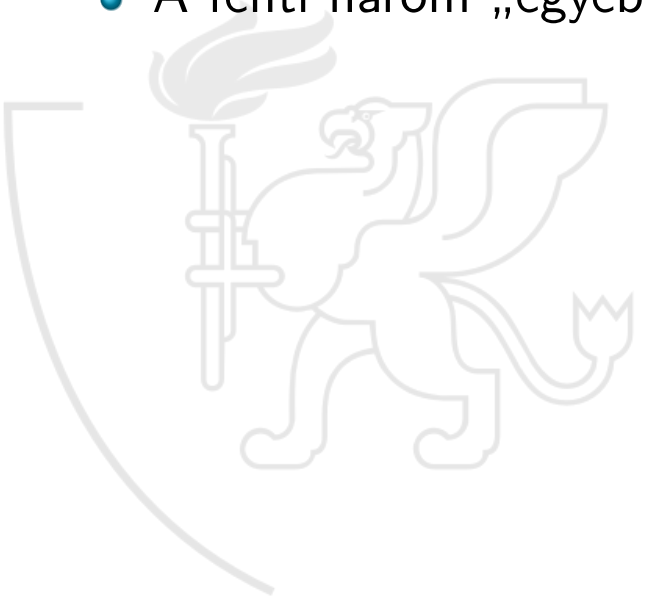
eltelt-fgv.c [1–26]

```c
/* Egy nap két időpontja között mennyi idő telt el.
 * 2007. Augusztus 30.  Gergely Tamás, gertom@inf.u-szeged.hu
 */

#include <stdio.h>

int eltelt_percek(int ora1, int perc1, int ora2, int perc2) {
    return 60 * ora2 + perc2 - (60 * ora1 + perc1);
}

int main() {
    int o1, p1;                         /* az első időpont */
    int o2, p2;                        /* a második időpont */
    int o, p;                             /* az eltelt idő */
    /* beolvasás */
    printf("Kérem az első időpontot óra perc formában\n");
    scanf("%d %d", &o1, &p1);
    printf("Kérem a második időpontot óra perc formában\n");
    scanf("%d %d", &o2, &p2);
    /* számítás */
    o = eltelt_percek(o1, p1, o2, p2) / 60;
    p = eltelt_percek(o1, p1, o2, p2) % 60;
    /* kiíratás */
    printf("Az eltelt idő: %d óra %d perc.\n", o, p);
    return 0;
}
```

Tartalom

Szelekciós vezérlések

- Szelekciós vezérlésről akkor beszélünk, amikor véges sok rögzített művelet közül véges sok feltétel alapján választjuk ki, hogy melyik művelet kerüljön végrehajtásra.
- Típusai:
  - Egyszerű szelekciós vezérlés
  - Többszörös szelekciós vezérlés
  - Esetkiválasztásos szelekciós vezérlés
  - A fenti három „egyébként” ággal



Háromszög osztályozása

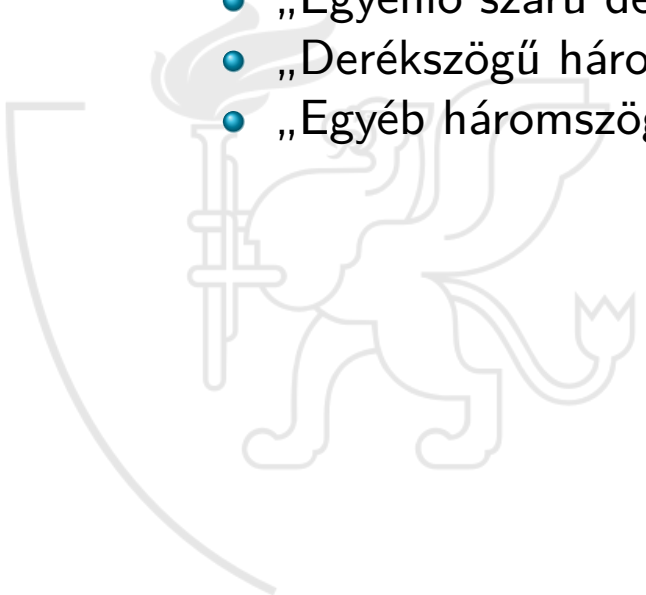
Problémafelvetés

- Problémafelvetés
  - Egy lakatosműhely egyik részlegében lemezekből vágnak ki különféle sokszögeket, köztük háromszögeket is. Többféle speciális gépük is van, amelyek bizonyos háromszögeket gyorsabban tudnak vágni, mint az „általános” vágógép. A gyártás optimalizálására a vezetőség szeretné, ha mindenféle háromszöget a neki leginkább megfelelő géppel vágnának ki. Van egy szabályos háromszögeket gyorsan vágó gép, egy-egy egyenlő szárú háromszögeket illetve derékszögű háromszögeket kicsit lassabban vágó gép, és egy általános de lassú vágógép. A háromszögeket a három oldaluk hosszával adják meg, és elképzelhetőek hibás adatok is.
  - A vezetőség szeretne egy programot, amely segít meghatározni a megfelelő (a feladathoz leggyorsabb) vágógépet azzal a kikötéssel, hogy az egyenlő szárú derékszögű háromszögek esetén a kettő megfelelő közül bármelyik gépet használhatják. Kellene tehát egy olyan program, ami megmondja, hogy milyen háromszöget határoz meg három valós szám, mint a háromszög három oldalhosszúsága?

Háromszög osztályozása

Specifikáció

- Specifikáció
  - A bemenő adat három valós szám, jelölje ezeket a, b és c.
  - A kimenő adat a három bemenő érték mint oldalhossz alapján a következő szövegek valamelyike:
    - „Nem háromszög”
    - „Szabályos háromszög”
    - „Egyenlő szárú háromszög”
    - „Egyenlő szárú derékszögű háromszög”
    - „Derékszögű háromszög”
    - „Egyéb háromszög”



Háromszög osztályozása

Algoritmustervezés

- Az osztályozás az alábbi feltételek alapján végezhető el:
  - Valamelyik oldal nem pozitív
    - $a \leq 0$ vagy $b \leq 0$ vagy $c \leq 0$
  - Nem alkotnak háromszöget
    - Feltéve, hogy a a hosszabbik oldal, $a \geq b + c$
  - Mindhárom oldal egyforma hosszúságú
    - $a = b = c$, azaz $a = b$ és $b = c$
  - Van legalább két egyforma oldal
    - $a = b$ vagy $b = c$ vagy $c = a$
  - A háromszög derékszögű
    - Feltéve, hogy a a hosszabbik oldal, $a^2 = b^2 + c^2$
- Látható, hogy az egyszerűbb feltételek érdekében az osztályozás előtt érdemes a legnagyobb értéket valamelyik, mondjuk az a változóba átmozgatni.
- Az osztályozás eredményét közvetlenül kiírjuk.

Egyszerű szelekciós vezérlés

- Egyszerű szelekció esetén egyetlen feltétel és egyetlen művelet van (ami persze lehet összetett).
- Legyen F logikai kifejezés, A pedig tetszőleges művelet. Az F feltételből és az A műveletből képzett egyszerű szelekciós vezérlés a következő vezérlési előírást jelenti:
  1. Értékeljük ki az F feltételt és folytassuk a 2.) lépéssel.
  2. Ha F értéke igaz, akkor hajtsuk végre az A műveletet és fejezzük be az összetett művelet végrehajtását.
  3. Egyébként, vagyis ha F értéke hamis, akkor fejezzük be az összetett művelet végrehajtását.

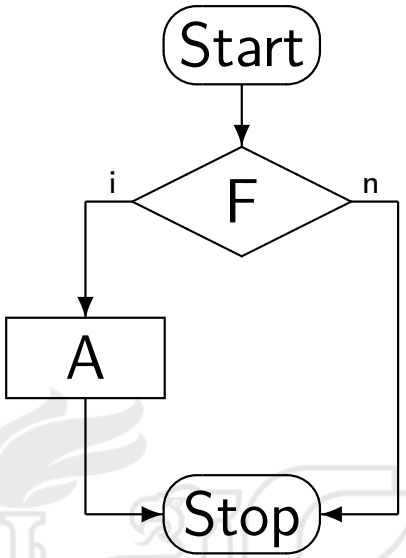
Egyszerű szelekciós vezérlés egyébként ággal

- A vezérlés bővíthető úgy, hogy a 3. pontban üres művelet helyett egy B műveletet hajtunk végre.
- Legyen F logikai kifejezés, A és B pedig tetszőleges művelet. Az F feltételből és az A és B műveletekből képzett egyszerű szelekciós vezérlés a következő vezérlési előírást jelenti:
  1. Értékeljük ki az F feltételt és folytassuk a 2.) lépéssel.
  2. Ha F értéke igaz, akkor hajtsuk végre az A műveletet és fejezzük be az összetett művelet végrehajtását.
  3. Egyébként, vagyis ha F értéke hamis, akkor hajtsuk végre a B műveletet és fejezzük be az összetett művelet végrehajtását.

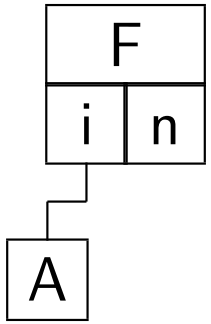
Egyszerű szelekciós vezérlés

Folyamatábra és szerkezeti ábra

- Folyamatábrája:



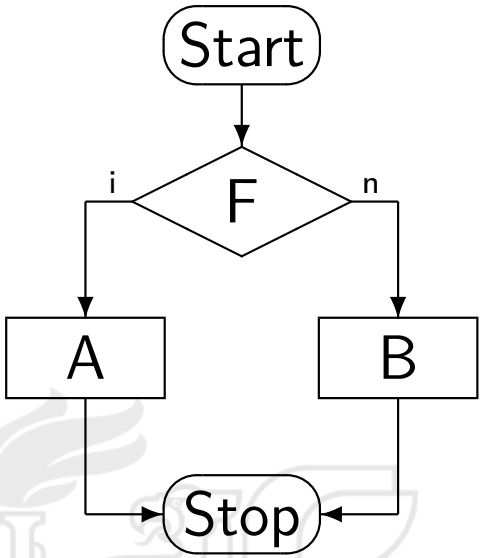
- Szerkezeti ábrája:



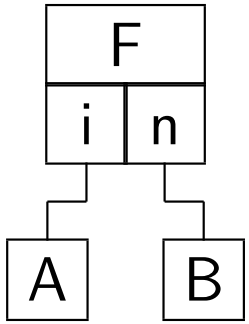
Egyszerű szelekciós vezérlés egyébként ággal

Folyamatábra és szerkezeti ábra

- Folyamatábrája:



- Szerkezeti ábrája:



Többszörös szelekciós vezérlés

- Ha több feltételünk és több műveletünk van, akkor többszörös szelekcióról beszélünk.
- Legyenek F_i logikai kifejezések, A_i pedig tetszőleges műveletek $1 \leq i \leq n$-re. Az F_i feltételekből és A_i műveletekből képzett többszörös szelekciós vezérlés a következő vezérlési előírást jelenti:
  1. Az F_i feltételek sorban történő kiértékelésével adjunk választ a következő kérdésre: Van-e olyan i $(1 \leq i \leq n)$, amelyre teljesül, hogy az F_i feltétel igaz és az összes F_j $(1 \leq j < i)$ feltétel hamis?
  2. Ha van ilyen i, akkor hajtsuk végre az A_i műveletet és fejezzük be az összetett művelet végrehajtását.
  3. Egyébként, vagyis ha minden F_i feltétel hamis, akkor fejezzük be az összetett művelet végrehajtását.

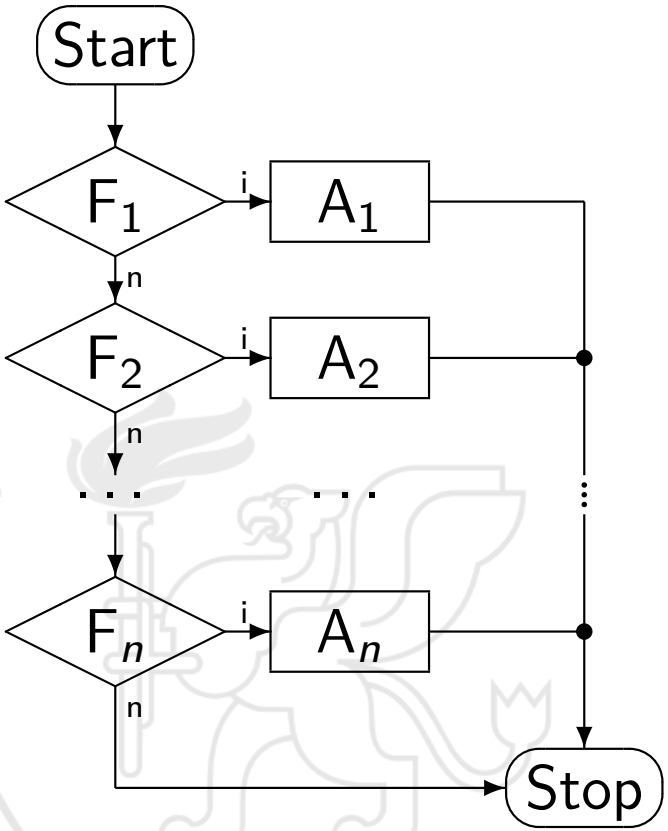
Többszörös szelekciós vezérlés egyébként ággal

- A többszörös szelekció is bővíthető egyébként ággal úgy, hogy egy nemüres B műveletet hajtunk végre a 3. lépésben.
- Legyenek F_i logikai kifejezések, A_i és B pedig tetszőleges műveletek $1 \leq i \leq n$-re. Az F_i feltételekből, A_i és B műveletekből képzett többszörös szelekciós vezérlés a következő vezérlési előírást jelenti:
  1. Az F_i feltételek sorban történő kiértékelésével adjunk választ a következő kérdésre: Van-e olyan i $(1 \leq i \leq n)$, amelyre teljesül, hogy az F_i feltétel igaz és az összes F_j $(1 \leq j < i)$ feltétel hamis?
  2. Ha van ilyen i, akkor hajtsuk végre az A_i műveletet és fejezzük be az összetett művelet végrehajtását.
  3. Egyébként, vagyis ha minden F_i feltétel hamis, akkor hajtsuk végre B-t és fejezzük be az összetett művelet végrehajtását.

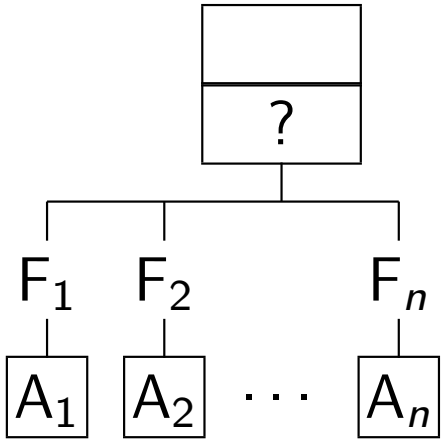
Többszörös szelekciós vezérlés

Folyamatábra és szerkezeti ábra

- Folyamatábrája:



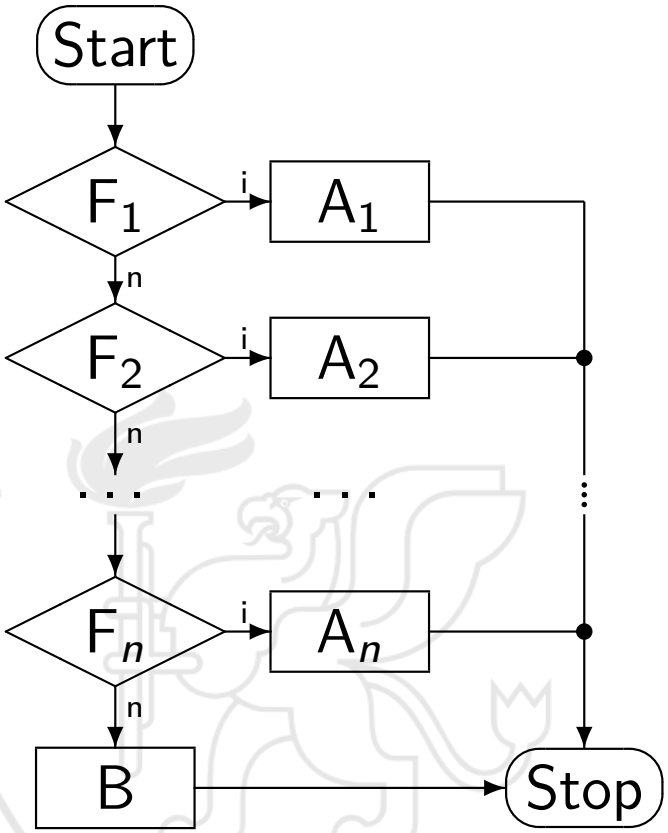
- Szerkezeti ábrája:



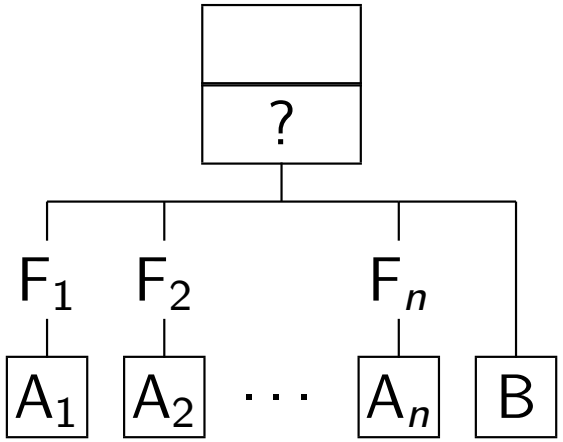
Többszörös szelekciós vezérlés egyébként ággal

Folyamatábra és szerkezeti ábra

- Folyamatábrája:



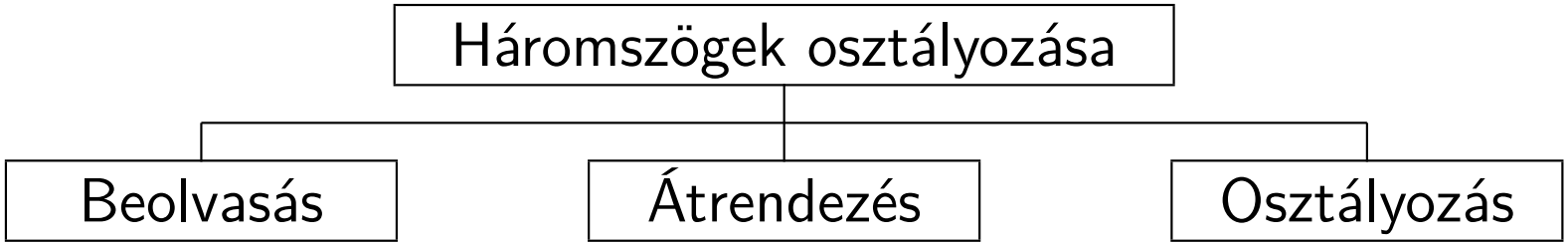
- Szerkezeti ábrája:



Háromszög osztályozása

Algoritmustervezés – Szerkezeti ábra 1.

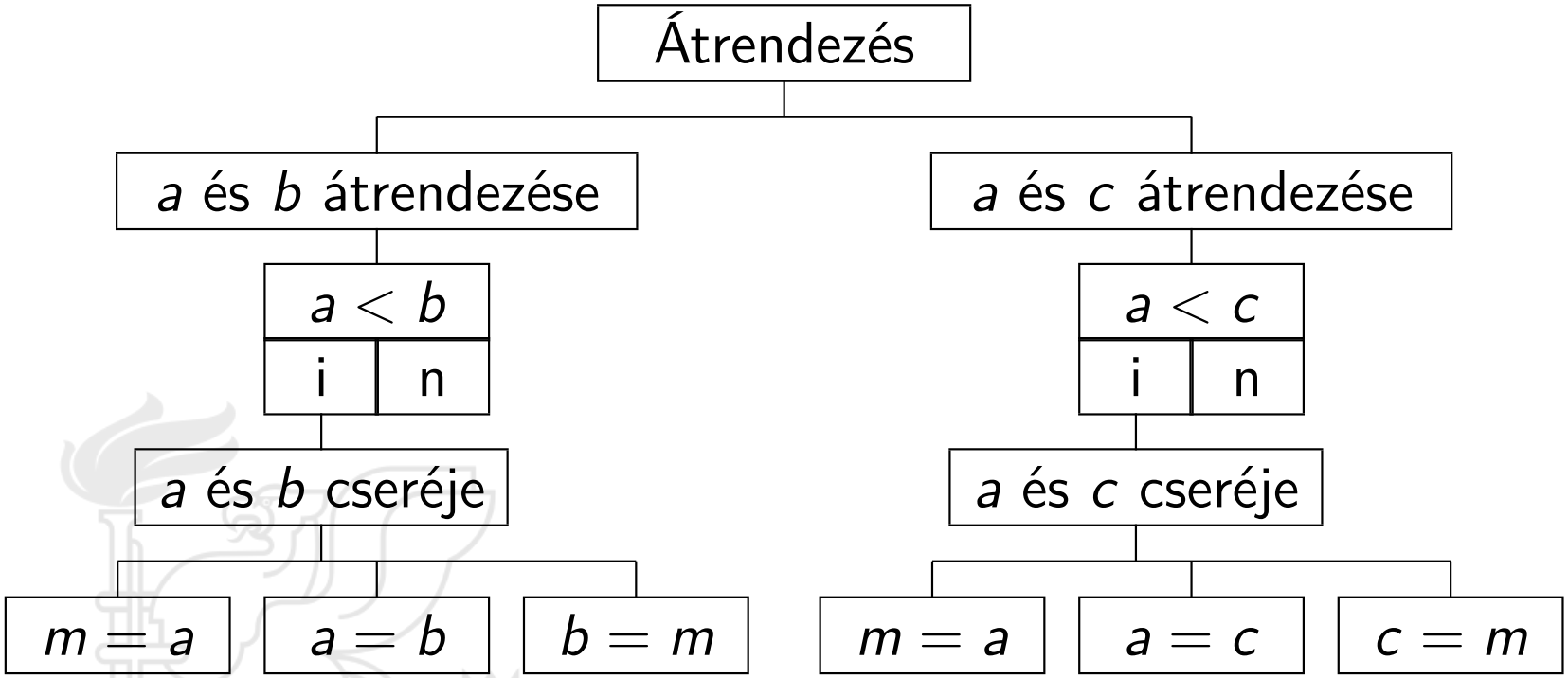
- A főprogram:



Háromszög osztályozása

Algoritmustervezés – Szerkezeti ábra 2.

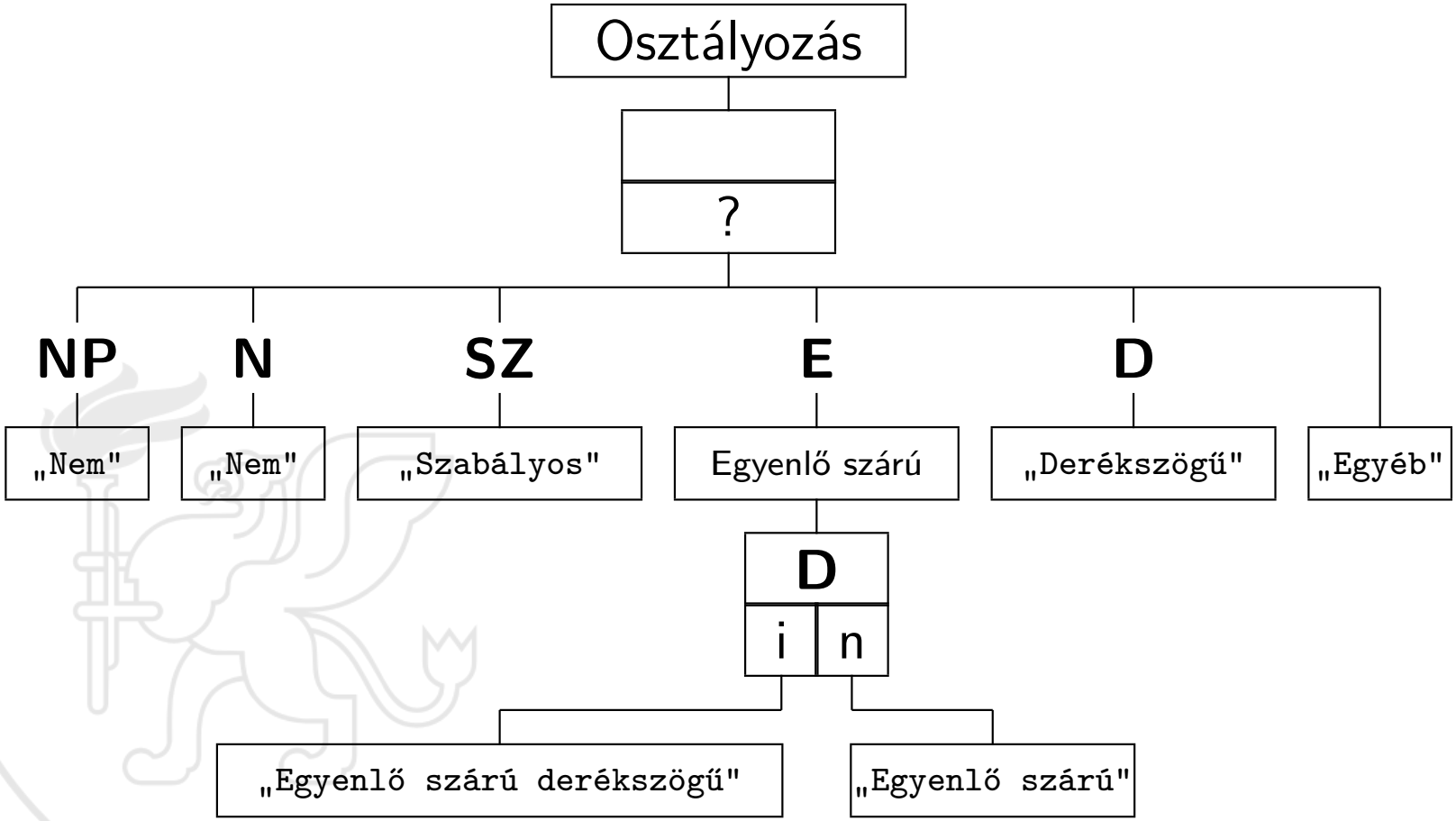
- Átrendezés:



Háromszög osztályozása

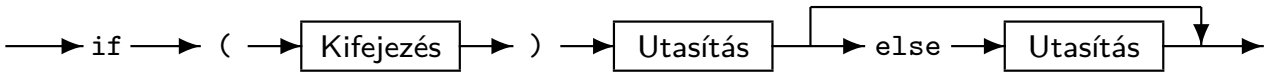
Algoritmustervezés – Szerkezeti ábra 3.

- Osztályozás:



Az `if` utasítás

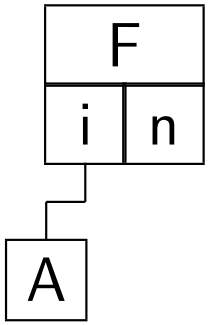
- Ha valamilyen feltétel alapján egyik vagy másik utasítást akarjuk végrehajtani.
- Az `if` utasítás szintaxisa C-ben



Egyszerű szelekciós vezérlés

Megvalósítás C nyelven

- Szerkezeti ábra:



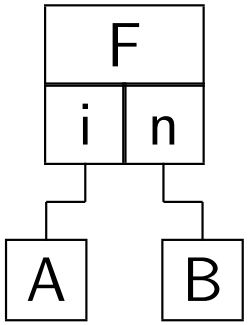
- Megvalósítás:

```
if (F) {
    A;
}
```

Egyszerű szelekciós vezérlés egyébként ággal

Megvalósítás C nyelven

- Szerkezeti ábra:



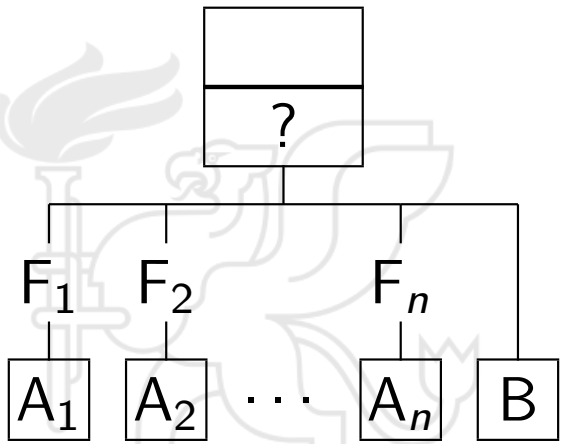
- Megvalósítás:

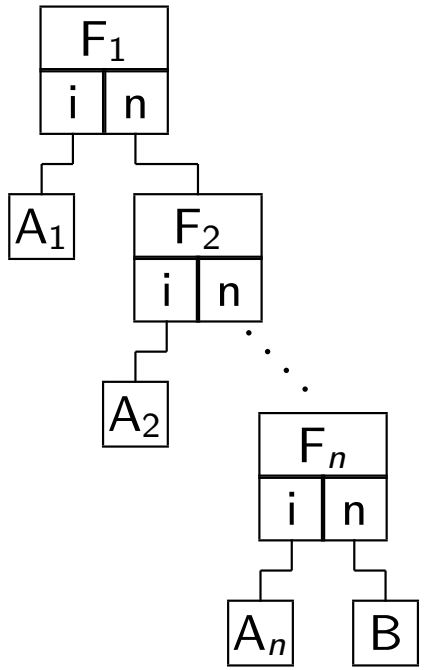
```
if (F) {
    A;
} else {
    B;
}
```

Többszörös szelekciós vezérlés

Megvalósítás C nyelven

- C nyelvben nincs külön utasítás a többszörös szelekció megvalósítására, ezért az egyszerű szelekció ismételt alkalmazásával kell azt megvalósítani.
- Ez azon az összefüggésen alapszik, hogy a többszörös szelekció levezethető egyszerű szelekciók megfelelő összetételével.

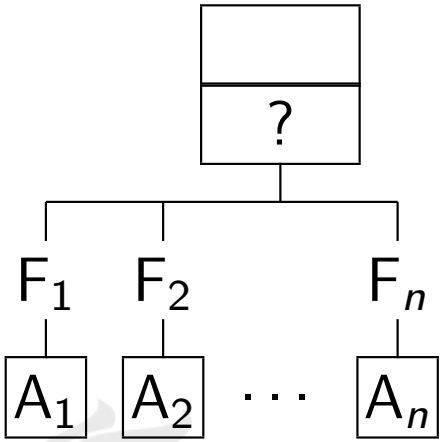




Többszörös szelekciós vezérlés

Megvalósítás C nyelven

- Szerkezeti ábra:



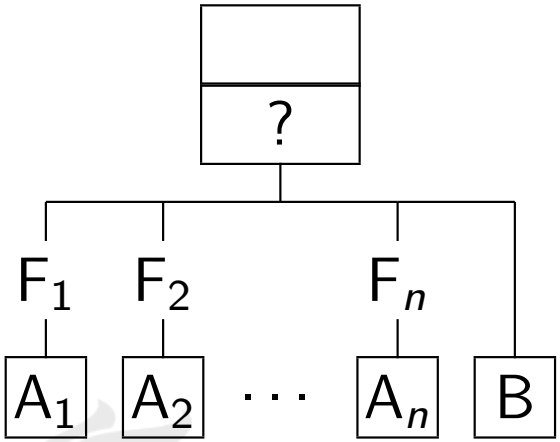
- Megvalósítás:

```
if (F1) {
    A1;
} else if (F2) {
    A2;
    ...
} else if (Fn) {
    An;
}
```

Többszörös szelekciós vezérlés egyébként ággal

Megvalósítás C nyelven

- Szerkezeti ábra:



- Megvalósítás:

```
if (F1) {
    A1;
} else if (F2) {
    A2;
    ...
} else if (Fn) {
    An;
} else {
    B;
}
```

Háromszög osztályozása [1/2]

haromszog.c [1–25]

```c
/* Milyen háromszöget határoz meg három pozitiv valós szám,
 *    mint a háromszög három oldalhosszúsága?
 * 1997. Október 13.  Dévényi Károly, devenyi@inf.u-szeged.hu
 */

#include <stdio.h>

int main() {
    double a, b, c;     /* a háromszög oldalhosszúságai */
    double m;           /* munkaváltozó a cseréhez      */

    printf("Kérem a három pozitív valós számot!\n");
    scanf("%lf%lf%lf", &a, &b, &c);

                              /* a,b,c átrendezése úgy, hogy a>=b,c legyen */
    if (a < b) {                                    /* a és b átrendezése */
        m = a;
        a = b;
        b = m;
    }
    if (a < c) {                                    /* a és c átrendezése */
        m = a;
        a = c;
        c = m;
    }
```

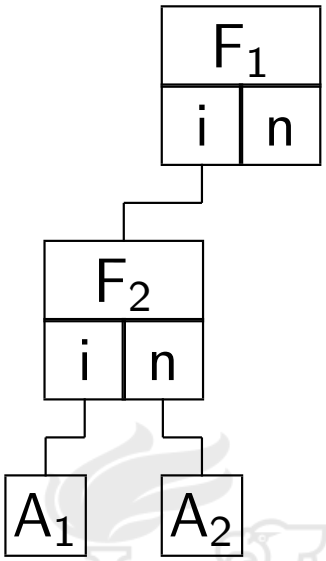
Háromszög osztályozása [2/2]

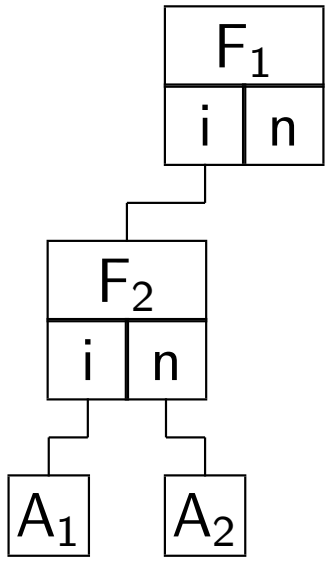
haromszog.c [27–46]

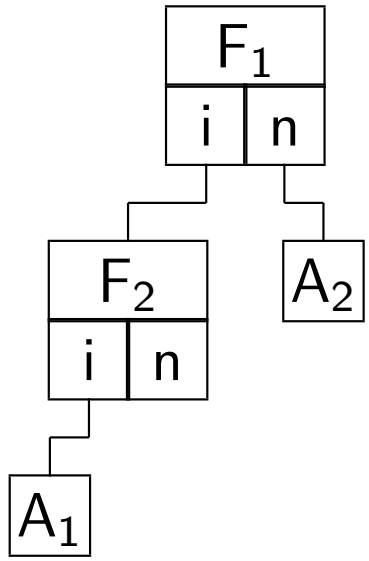
```
27      if (b <= 0 || c <= 0) {
28          printf("Nem háromszög!\n");                    /* 1. alternatíva */
29      } else if (a >= b + c) {
30          printf("Nem háromszög!\n");                    /* 2. alternatíva */
31      } else if (a == b && b == c) {
32          printf("Szabályos háromszög.\n");              /* 3. alternatíva */
33      } else if (a == b || b == c || a == c) {
34          if (a * a == b * b + c * c) {                  /* 4. alternatíva */
35              printf("Egyenlő szárú derékszögű háromszög.\n");
36          } else {
37              printf("Egyenlő szárú háromszög.\n");
38          }
39      } else if (a * a == b * b + c * c) {
40          printf("Derékszögű háromszög.\n");             /* 5. alternatíva */
41      } else {
42          printf("Egyéb háromszög.\n");                      /* egyébként */
43      }
44                                          /* vége a többszörös szelekciónak */
45      return 0;
46  }
```

if utasítások ismételt alkalmazása

- Figyeljünk oda a megfelelő zárójelezésre!








```
if (F1) {
    if (F2)
        A1;
    else
        A2;
}
```

```
if (F1)
    if (F2)
        A1;
else
    A2;
```

```
if (F1) {
    if (F2)
        A1;
} else
    A2;
```

Dátum helyességének eldöntése

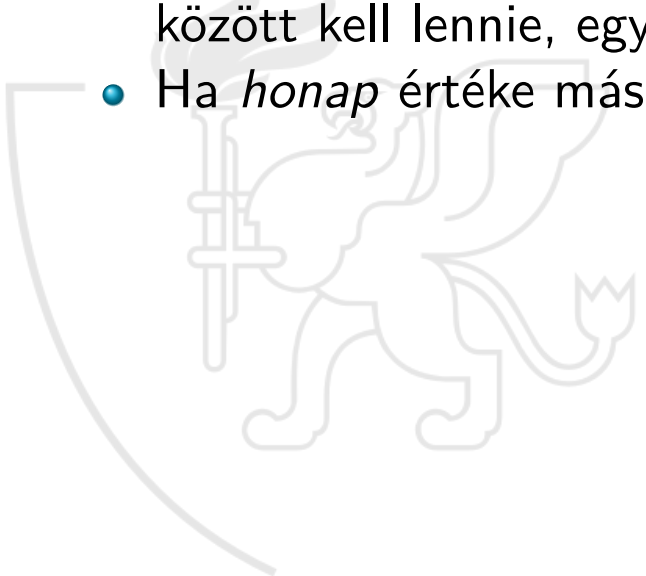
Problémafelvetés és specifikáció

- Problémafelvetés
  - A munkaügyi hatóság ellenőrzéseket tart építkezéseken, ahol a munkanaplókat is ellenőrzik. Mivel tapasztalat, hogy az ellenőrzés hírére pánikszerűen kitöltött naplókban sok rossz dátum is szerepel, szükségük lenne egy olyan programra, amely mintegy „előszűri" az adatokat, és a nyilvánvalóan hibás dátumokat jelzi.
  - Vagyis kellene egy program, ami eldönti, hogy egy dátumként megadott számpár helyes dátum-e?
- Specifikáció
  - Az input egy *honap nap* egész számpár által megadott dátum.
  - Az output „A dátum helyes" vagy „A dátum nem helyes" szövegek valamelyike attól függően, hogy a *honap nap* dátum hónap napként helyes-e vagy sem.
    - Szökőévekkel nem foglalkozunk, a február 28 napos.

Dátum helyességének eldöntése

Algoritmustervezés

- Először beolvassuk a két számot (*honap nap*).
- Eldöntjük, hogy helyes-e a dátum.
  - Ha *honap* értéke 2, a *nap* értékének 1 és 28 közé kell esnie, különben a dátum nem helyes.
  - Ha *honap* értéke 4, 6, 9 vagy 11, a *nap* értéke 1 és 30 közé kell, hogy essen, másként a dátum nem helyes.
  - Ha *honap* értéke 1, 3, 5, 7, 8, 10 vagy 12, a *nap* értékének 1 és 31 között kell lennie, egyébként a dátum nem jó.
  - Ha *honap* értéke más, a dátum nem helyes.



Esetkiválasztásos szelekciós vezérlés

- Ha a többszörös szelekciós vezérlésben minden F_i feltételünk $K \in H_i$ alakú, akkor esetkiválasztásos szelekcióról beszélünk.
- Legyen K egy adott típusú kifejezés, legyenek H_i ilyen típusú halmazok, A_i pedig tetszőleges műveletek $1 \leq i \leq n$-re. A K szelektor kifejezésből, H_i kiválasztó halmazokból és A_i műveletekből képzett esetkiválasztásos szelekciós vezérlés a következő vezérlési előírást jelenti:
  1. Értékeljük ki a K kifejezést és folytassuk a 2.) lépéssel.
  2. Adjunk választ a következő kérdésre: Van-e olyan i $(1 \leq i \leq n)$, amelyre teljesül, hogy a $K \in H_i$, és $K \notin H_j$ $(1 \leq j < i)$?
  3. Ha van ilyen i, akkor hajtsuk végre az A_i műveletet és fejezzük be az összetett művelet végrehajtását.
  4. Egyébként, vagyis ha K nem eleme egyetlen H_i halmaznak sem, akkor fejezzük be az összetett művelet végrehajtását.

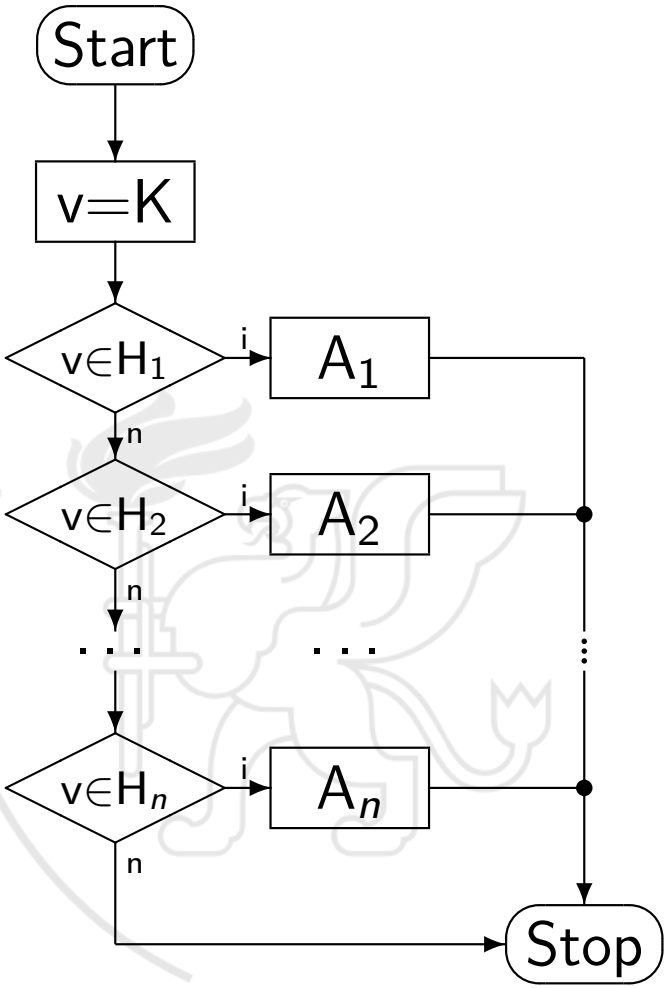
Esetkiválasztásos szelekciós vezérlés egyébként ággal

- A többszörös szelekció is bővíthető egyébként ággal úgy, hogy egy nemüres B műveletet hajtunk végre a 4. lépésben.
- Legyen K egy adott típusú kifejezés, legyenek H_i ilyen típusú halmazok, A_i és B pedig tetszőleges műveletek $1 \leq i \leq n$-re. A K szelektor kifejezésből, H_i kiválasztó halmazokból és A_i és B műveletekből képzett esetkiválasztásos szelekciós vezérlés a következő vezérlési előírást jelenti:
  1. Értékeljük ki a K kifejezést és folytassuk a 2.) lépéssel.
  2. Adjunk választ a következő kérdésre: Van-e olyan i $(1 \leq i \leq n)$, amelyre teljesül, hogy a $K \in H_i$, és $K \notin H_j$ $(1 \leq j < i)$?
  3. Ha van ilyen i, akkor hajtsuk végre az A_i műveletet és fejezzük be az összetett művelet végrehajtását.
  4. Egyébként, vagyis ha K nem eleme egyetlen H_i halmaznak sem, akkor hajtsuk végre B-t és fejezzük be az összetett művelet végrehajtását.

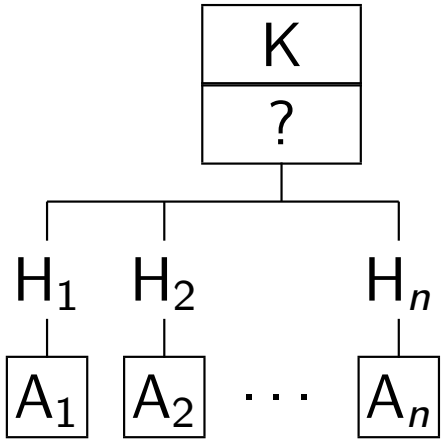
Esetkiválasztásos szelekciós vezérlés

Folyamatábra és szerkezeti ábra

- Folyamatábrája:



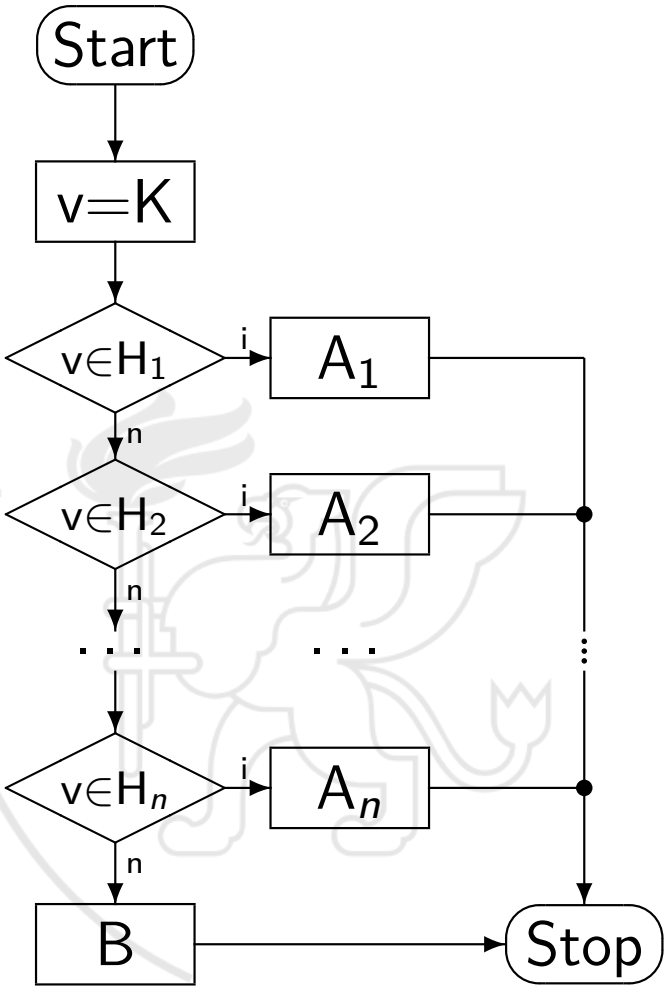
- Szerkezeti ábrája:



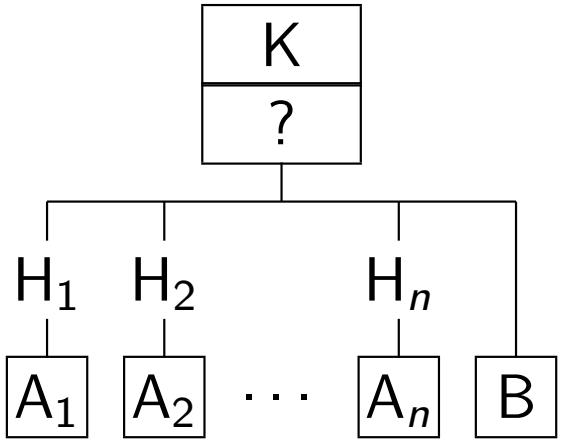
Esetkiválasztásos szelekciós vezérlés egyébként ággal

Folyamatábra és szerkezeti ábra

- Folyamatábrája:



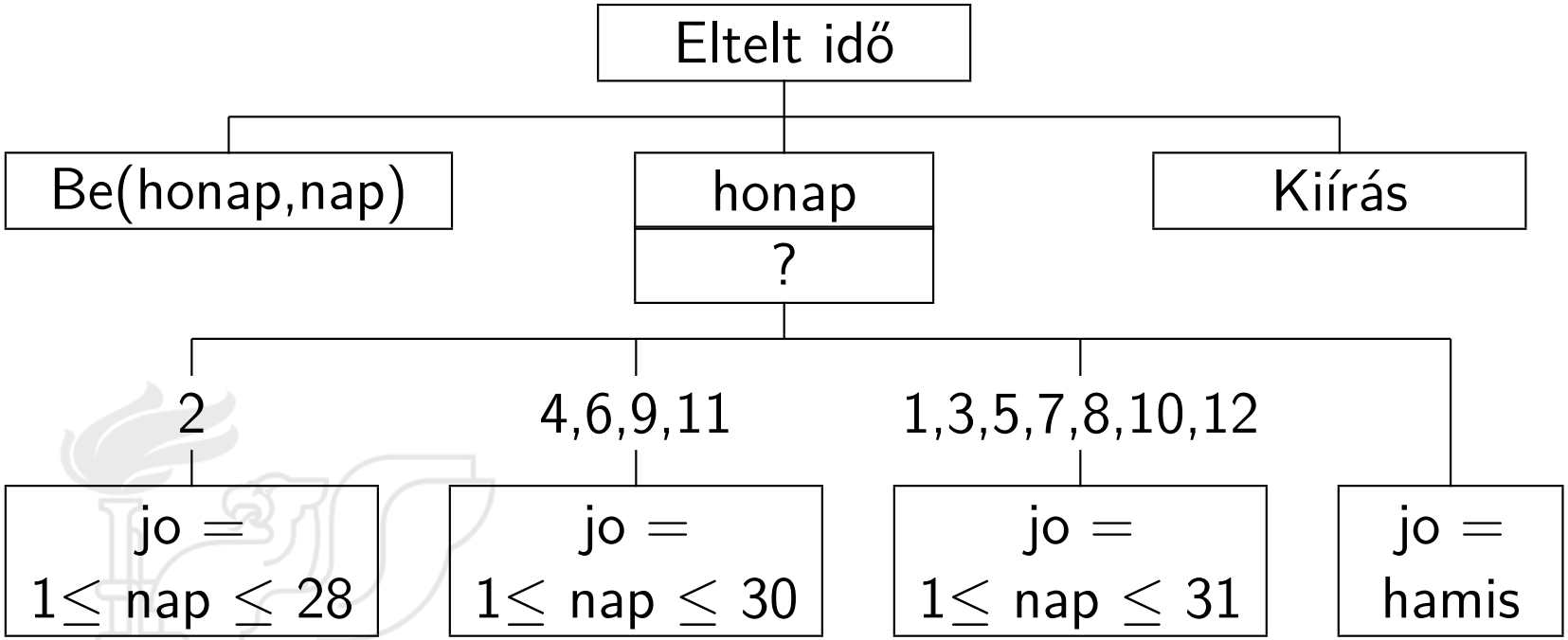
- Szerkezeti ábrája:



Dátum helyességének eldöntése

Algoritmustervezés – Szerkezeti ábra

- A dátum helyessége probléma megoldásának szerkezeti ábrája:



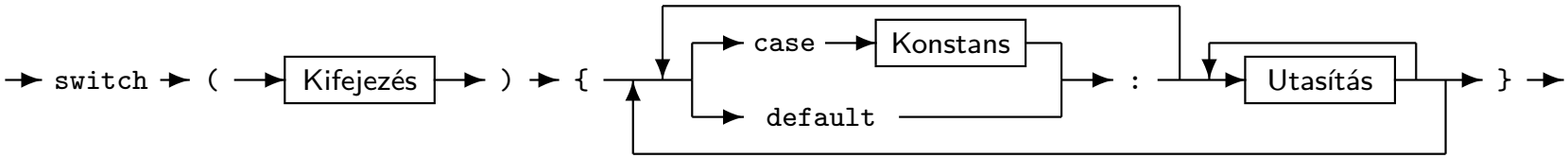
Esetkiválasztásos szelekciós vezérlés

Halmazok megadása

- A kiválasztó halmazok megadása az esetkiválasztásos szelekció kritikus pontja.
- Algoritmusok tervezése során bármilyen effektív halmazmegadást használhatunk, azonban a tényleges megvalósításkor csak a választott programozási nyelv eszközeit alkalmazhatjuk.

A `switch` utasítás

- Ha egy kifejezés értéke alapján többféle utasítás közül kell választanunk, a `switch` utasítást használhatjuk.
  - Megadhatjuk, hogy hol kezdődjön és meddig tartson az utasítás-sorozat végrehajtása.
- A `switch` utasítás szintaxisa C-ben



A `switch` utasítás

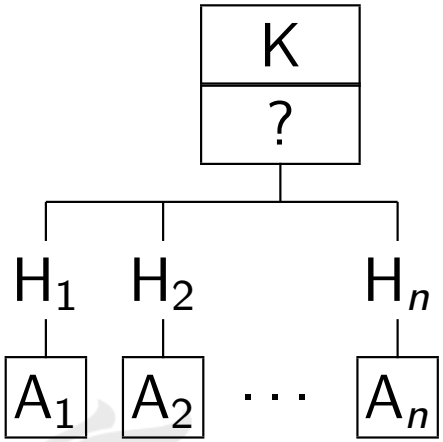
Tulajdonságok

- A szelektor kifejezés és a konstansok típusának meg kell egyeznie.
- Egy konstans legfeljebb egy `case` mögött és a `default` kulcsszó is legfeljebb egyszer szerepelhet egy `switch` utasításban.
- A `default` cimke olyan, mintha a szelektor kifejezés lehetséges értékei közül minden olyat felsorolnánk, ami nem szerepel `case` mögött az adott `switch`-ben.
- A cimkék (beleértve a `default`-ot is) sorrendje tetszőleges lehet, az nem befolyásolja, hogy a szelektor kifejezés melyik cimkét választja.
- A szelektor kifejezés értékétől csak az függ, hogy melyik helyen kezdjük el végrehajtani a `switch` magját. Ha a végrehajtás elkezdődik, akkor onnantól kezdve az első `break` (vagy `return`) utasításig, vagy a `switch` végéig sorban hajtódnak végre az utasítások. Ebben a fázisban a további `case` és `default` cimkéknek már nincs jelentőssége.

Esetkiválasztásos szelekciós vezérlés

Megvalósítás C nyelven

- Szerkezeti ábra:



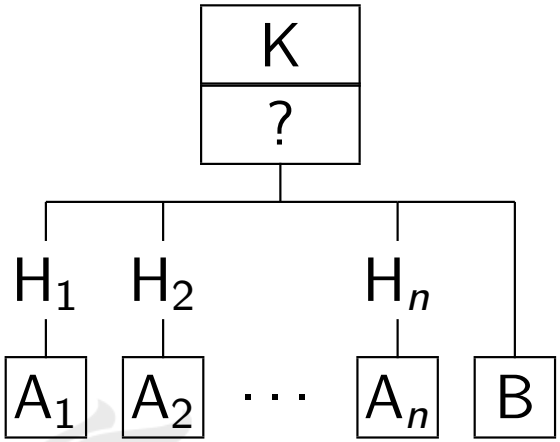
- Megvalósítás:

```
switch (K) {
    case H1:
        A1;
        break;
    ...
    case Hn:
        An;
        break;
}
```

Esetkiválasztásos szelekciós vezérlés egyébként ággal

Megvalósítás C nyelven

- Szerkezeti ábra:



- Megvalósítás:

```
switch (K) {
    case H1:
        A1;
        break;
    ...
    case Hn:
        An;
        break;
    default:
        B;
        break;
}
```

Esetkiválasztásos szelekciós vezérlés

Megvalósítás C nyelven

- A H_i halmazok elemszáma tetszőleges lehet, a `case`-ek után viszont csak egy-egy érték állhat.
- Kihasználhatjuk viszont a `switch` működési elvét, miszerint a cimkék csak a belépési pontot határozák meg. Ha $H_i = \{x_{i,1}, x_{i,2}, \ldots, x_{i,n_i}\}$, akkor az adott ág lekódolható a

```
case x_{i,1}:
case x_{i,2}:
...
case x_{i,n_i}:
    A_i;
    break;
```

kódrészlettel.

Logikai adattípus C nyelven

- A C nyelvnek csak a C99 szabvány óta része a logikai (`_Bool`) típus (melynek értékkészlete a $\{0, 1\}$ halmaz), de azért logikai értékek persze előtte is keletkeztek.
- A műveletek eredményeként keletkező logikai hamis értéket a 0 egész érték reprezentálja, és a 0 egész érték logikai értékként értelmezve hamisat jelent.
- A műveletek eredményeként keletkező logikai igaz értéket az 1 egész érték reprezentálja, de bármely 0-tól különböző egész érték logikai értékként értelmezve igazat jelent.
- A logikai és egész értékek C-ben teljesen konvertibilisek (a fenti konverziók szerint), így logikai értékeket egész típusú változókban is tudunk tárolni (sőt, C99 előtt abban kellett).
- A C99 szabvány bevezette az `stdbool.h` header-t is, ami definiálja a „jobban kinéző” `bool` típust, valamint a `false` és `true` literálokat is.

Dátum helyességének eldöntése

Algoritmustervezés – Változók

- Az algoritmustervezés során használtunk egy „jo” nevű változót, amely logikai értéket tárolt.
- Ebben a megvalósításban ezt a változót `int` típusúként deklaráljuk.

Dátum helyességének eldöntése [1/3]

datum.c [1–12]

```c
/* Eldöntendő, hogy egy dátumként megadott számpár helyes dátum-e?
 * 1997. Október 4.  Dévényi Károly, devenyi@inf.u-szeged.hu
 */

#include <stdio.h>

int main() {
    int honap, nap;
    int jo;                                   /* a Boolean érték tárolására */

    printf("Kérem␣a␣dátumot␣(hónap,␣nap)␣!\n");
    scanf("%d%d", &honap, &nap);
```

Dátum helyességének eldöntése [2/3]

datum.c [14–36]

```
14      switch (honap) {
15      case 2:
16          jo = (1 <= nap && nap <= 28);
17          break;
18      case 4:
19      case 6:
20      case 9:
21      case 11:
22          jo = (1 <= nap && nap <= 30);
23          break;
24      case 1:
25      case 3:
26      case 5:
27      case 7:
28      case 8:
29      case 10:
30      case 12:
31          jo = (1 <= nap && nap <= 31);
32          break;
33      default:
34          jo = 0;
35          break;
36      } /* switch */
```

Dátum helyességének eldöntése [3/3]

datum.c [37–44]

```
37                                                          /* Kiíratás  */
38      printf("A␣dátum␣");
39      if (!jo) {                                      /* Ezt másképpen szokták */
40          printf("nem␣");
41      }
42      printf("helyes.\n");
43      return 0;
44  }
```

Feltételes kifejezés

- A feltételes operátor a C nyelv egyetlen háromoperandusú művelete. A K&R könyv feltételes kifejezésnek említi.

```
kifejezés1 ? kifejezés2 : kifejezés3
```

- Először a `kifejezés1` kerül kiértékelésre, ha ez
  - igaz (nem 0), a kifejezés értéke `kifejezés2` lesz,
  - hamis (0), a kifejezés értéke `kifejezés3` lesz.

Feltételes kifejezés

- Az előző program kiíratása

```
39      printf("A␣dátum␣");
40      if (!jo) {
41          printf("nem␣");
42      }
43      printf("helyes.\n");
```

lerövidíthető

```
39      printf("A␣dátum");
40      printf(jo ? "␣" : "␣nem␣");
41      printf("helyes.\n");
```

vagy

```
39      printf(jo ?
40          "A␣dátum␣helyes.\n" :
41          "A␣dátum␣nem␣helyes.\n");
```

alakban.

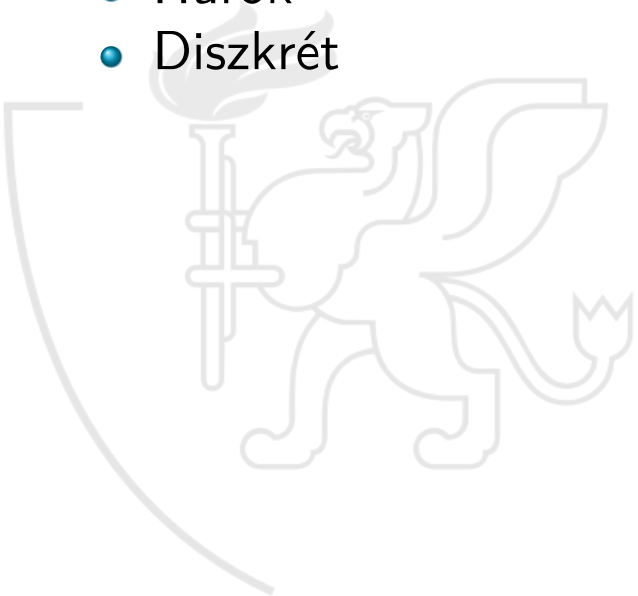
C műveletek prioritása

műveletek	asszoc.	C operátorok
egyoperandusú postfix, elemkiválasztás, mezőkiválasztás, függvényhívás	→	`x++`, `x--`, `[]`, `.`, `->`, `f()`
egyoperandusú prefix, méret, típuskényszerítés	←	`+`, `-`, `++x`, `--x`, `!`, `~`, `&`, `*`, `sizeof`, `(type)`
multiplikatív	→	`*`, `/`, `%`
additív	→	`+`, `-`
bitléptetés	→	`<<`, `>>`
kisebb-nagyobb relációs	→	`<=`, `>=`, `<`, `>`
egyenlő-nem egyenlő relációs	→	`==`, `!=`
bitenkénti 'és'	→	`&`
bitenkénti 'kizáró vagy'	→	`^`
bitenkénti 'vagy'	→	`|`
logikai 'és'	→	`&&`
logikai 'vagy'	→	`||`
feltételes	←	`?:`
értékadó	←	`=`, `+=`, `-=`, `*=`, `/=`, `%=` `>>=`, `<<=`, `&=`, `^=`, `|=`
szekvencia	→	`,`

Tartalom

Ismétléses vezérlések

- Ismétléses vezérlésen olyan vezérlési előírást értünk, amely adott műveletnek adott feltétel szerinti ismételt végrehajtását írja elő.
- Az ismétlési feltétel szerint ötféle ismétléses vezérlést különböztetünk meg:
  - Kezdőfeltételes
  - Végfeltételes
  - Számlálásos
  - Hurok
  - Diszkrét



Ismétléses vezérlések

- Az algoritmustervezés során a leginkább megfelelő ismétléses vezérlési formát használjuk, függetlenül attól, hogy a megvalósításra használt programozási nyelvben közvetlenül megvalósítható-e ez a vezérlési mód.
- Ismétléses vezérlés képzését ciklusszervezésnek is nevezik, így az ismétlésben szereplő műveletet ciklusmagnak hívjuk.

Számsorozat jellemzői

Problémafelvetés és specifikáció

- Problémafelvetés
  - A globális felmelegedés a nyakunkon van, ez egyre inkább érezhető. Vannak viszont, akiknek ez az „érzés” nem elegendő, csak a bizonyítékoknak hisznek. (És vannak, akik azoknak sem, de az már messze vezet.) A meteorológusok a több, mint száz éve rendszeresen naponta többször mért adatokból szeretnének statisztikákat összeállítani, amik a napi, heti, havi, negyedéves minimum, maximum és átlaghőmérsékleteket mutatják, hogy ezek változásával mutassák ki a jellemző trendeket. Egy-egy ilyen adathármas előállításához sok-sok (adott intervallumba eső) hőmérsékleti értéket kell feldolgozni, ezért ehhez számítógépes segítséget kérnek.
  - Kellene egy program, amely meghatározza egy valós számsorozat legkisebb és legnagyobb elemét, valamint a sorozat átlagát!

Számsorozat jellemzői

Problémafelvetés és specifikáció

- Specifikáció
  - A probléma inputja egy valós számokból álló sorozat.
  - Az input számsorozat végét egy végjel fogja jelezni, amit a felhasználó ad meg inputként, nyilván a számsorozat előtt.
  - Az output a sorozat legkisebb és legnagyobb eleme, valamint az átlaga.

Számsorozat jellemzői

Algoritmustervezés 1.

- Elsőre talán az tűnne a legegyszerűbb megoldásnak, ha beolvasnánk az összes számot, majd ezek között megkeresnénk a legkisebbet, legnagyobbat, majd kiszámolnánk az átlagot.
- De hogyan is működne a három részalgoritmus?
  - Minimum keresése:
    - Legyen kezdetben a *minimum* az első elem.
    - Vizsgáljuk sorban a sorozat elemeit. Ha a vizsgált elem kisebb a *minimum*nál, akkor ez lesz az új *minimum*. Az összes elem feldolgozása után a *minimum* a teljes sorozat minimuma lesz.
  - A maximális elem keresése ehhez teljesen hasonlóan megy.
  - Az átlagszámításhoz össze kell adni a sorozat összes elemét, majd a végén el kell osztani a sorozat elemeinek számával.
    - Legyen az *összeg* és az *elemszám* kezdetben nulla.
    - Vizsgáljuk sorban a sorozat elemeit. Minden elemnél az elem értékét adjuk hozzá az *összeg*hez, és növeljük eggyel az *elemszám*ot.

Számsorozat jellemzői

Algoritmustervezés 2.

- Ha végiggondoljuk, a három részalgoritmus összevonható.
- A soron következő elem feldolgozásával ki tudjuk számolni az eddigi sorozat minimumát, maximumát, összegét és elemszámát.
- Ha ezt megtettük, a sorozat ezen elemére nincs többé szükség, azt nem kell eltárolni.
  - Nem lesz szükségünk tömbre (aminek nem is tudnánk előre a méretét), kevesebb memóriát használ a programunk.
  - Olyan sorozatokat is fel tudunk dolgozni, amelyek nem férnének el a memóriában.
- Megjegyeznénk, hogy egy sorozat minimumának/maximumának kiválasztása, illetve elemeiből egyetlen érték előállítása úgynevezett *programozási tételek*, vagyis nagyon alapvető algoritmusok. Ezeket konkrétan *szélsőérték-kiválasztás*nak és *összegzés*nek nevezzük.

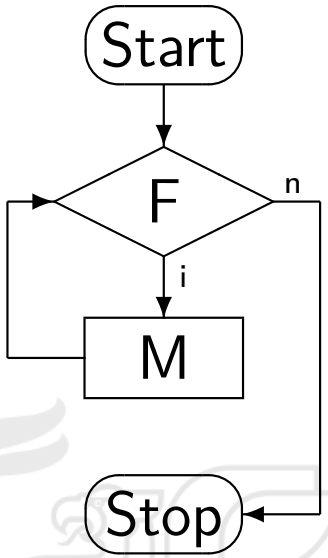
Kezdőfeltételes ismétléses vezérlés

- Kezdőfeltételes vezérlésről akkor beszélünk, ha a ciklusmag (ismételt) végrehajtását egy belépési (ismétlési) feltételhez kötjük.
- Legyen F logikai kifejezés, M pedig tetszőleges művelet. Az F ismétlési feltételből és az M műveletből (a ciklusmagból) képzett kezdőfeltételes ismétléses vezérlés a következő vezérlési előírást jelenti:
  1. Értékeljük ki az F feltételt és folytassuk a 2.) lépéssel.
  2. Ha F értéke hamis, akkor az ismétlés és ezzel együtt az összetett művelet végrehajtása befejeződött.
  3. Egyébként, vagyis ha az F értéke igaz, akkor hajtsuk végre az M műveletet, majd folytassuk az 1.) lépéssel.

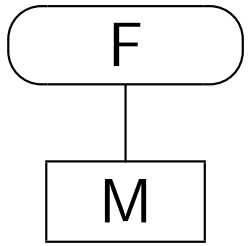
Kezdőfeltételes ismétléses vezérlés

Folyamatábra és szerkezeti ábra

- Folyamatábrája:



- Szerkezeti ábrája:

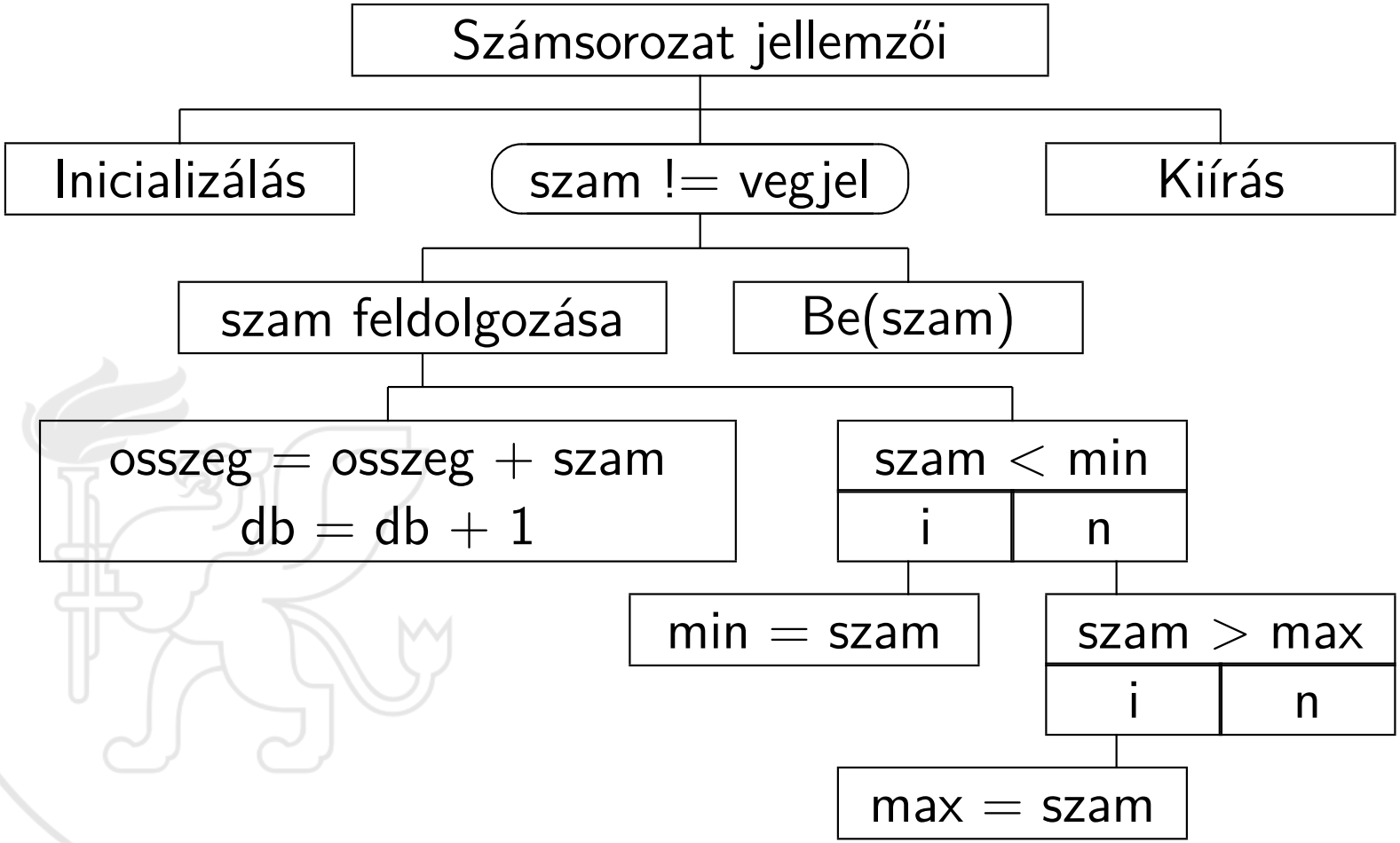


- Ha az *F* értéke kezdetben hamis, *M* egyszer sem kerül végrehajtásra.
- Ha az *F* értéke kezdetben igaz, és az *M* művelet nincs hatással *F*-re, akkor végtelen ciklust kapunk.

Számsorozat jellemzői

Algoritmustervezés – Szerkezeti ábra 1.

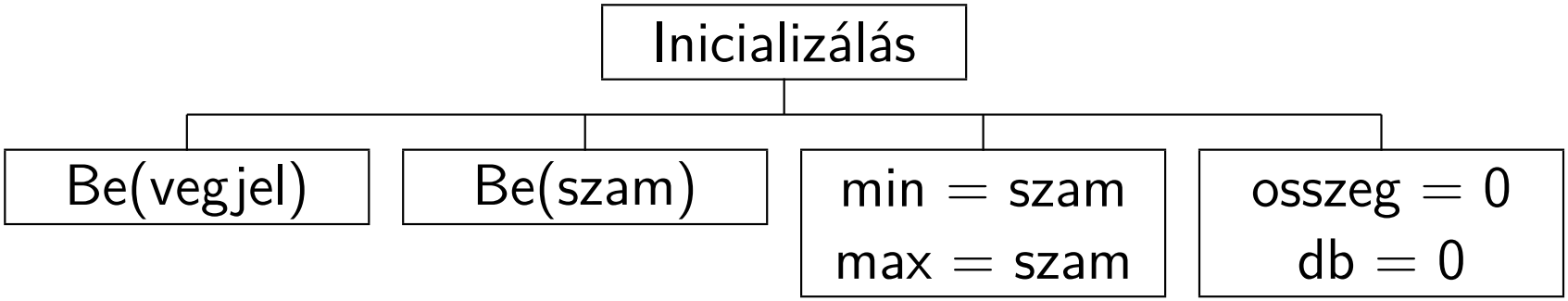
- A főprogram:



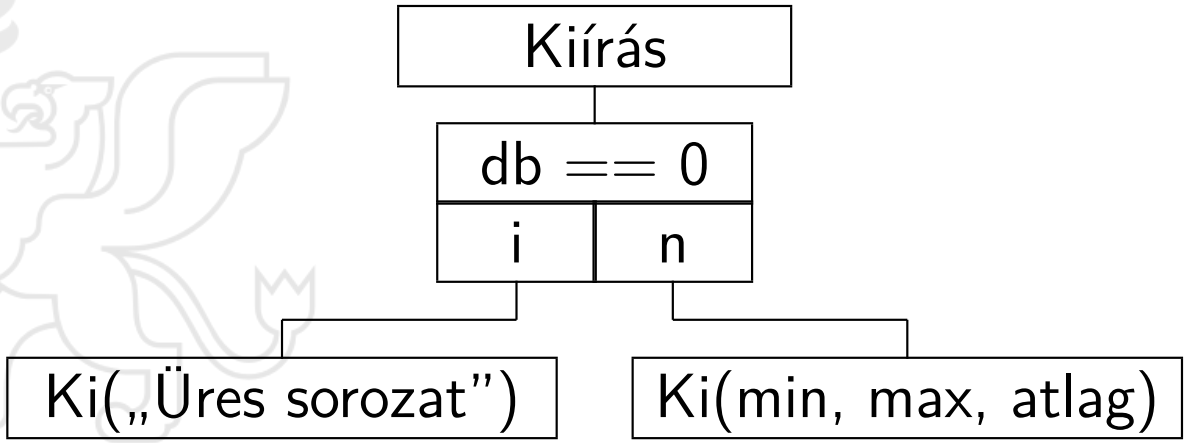
Számsorozat jellemzői

Algoritmustervezés – Szerkezeti ábra 2.

- Inicializálás:



- Kiírás:



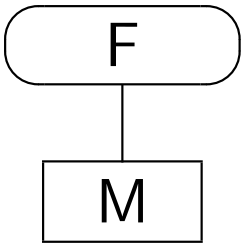
A `while` utasítás

- Ha valamilyen műveletet mindaddig végre kell hajtani, amíg egy feltétel igaz a `while` utasítás (is) használható.
  - A művelet végrehajtása nem szükséges a feltétel kiértékeléséhez.
  - A feltétel ellenőrzése a művelet előtt történik, így ha a feltétel kezdetben hamis volt, a műveletet egyszer sem hajtjuk végre.
- A `while` utasítás szintaxisa C-ben

Kezdőfeltételes ismétléses vezérlés

Megvalósítás C nyelven

- Szerkezeti ábrája:



- Megvalósítása:

```
while (F) {
    M;
}
```

A C értékadó műveletei

- A C nyelv egyik jó tulajdonsága, hogy különféle operátorai és konstrukciói által tömör kód írását is lehetővé teszi.
- Mielőtt elhamarkodottan lekódolnánk az előző szerkezeti ábrán található programot, ismerkedjünk meg a C nyelv néhány ilyen konstrukciójával.
  - Inkrementáló és dekrementáló műveletek
  - Egyéb műveletekkel kombinált értékadó műveletek

Inkrementáló és dekrementáló műveletek

- Egy program során gyakran előfordul, hogy egy egész változó értékét eggyel kell növelni vagy csökkenteni közvetlenül a felhasználása előtt vagy után.
- A C nyelv tartalmaz két operátort, amelyekkel változók inkrementálhatók és dekrementálhatók.
  - A ++ inkrementáló operátor az operandusa értékét 1-gyel növeli.
  - A -- dekrementáló operátor az operandusa értékét 1-gyel csökkenti.
  - Ezek használhatóak prefix (`++i`, `--i`) és postfix (`i++`, `i--`) alakban is.
    - Prefix alakban a művelet mint kifejezés értéke az operandus új értéke lesz.
    - Postfix alakban a művelet mint kifejezés értéke az operandus régi értéke lesz (de az operandus értéke a művelet után az új, módosított érték lesz).
  - A prefix és postfix használat között akkor van különbség, ha nem csak a művelet inkrementáló/dekrementáló tulajdonságát, hanem a kifejezés értékét is felhasználjuk.

Inkrementáló és dekrementáló műveletek

Példa

- Legyen `i` értéke 5, ekkor az

```
x = ++i;
```

utasítás az `x` értékét `i` új értékére, vagyis 6-ra állítja, az

```
x = i++;
```

utasítás viszont az `x` értékét `i` eredeti értékére, azaz 5-re. Az `i` értéke a művelet elvégzése után mindkét esetben 6 lesz.

- Ezek az operátorok értékadó utasításnak számítanak, így csak változókra (l-value) alkalmazhatók; az olyan kifejezés, mint például az

```
(i + j)++
```

nem megengedett!

A C értékadó műveletei

- C-ben a

```
v = v ⊙ e
```

alakú értékadásokat

```
v ⊙= e
```

alakban rövidíthetjük, ahol a $\odot$ tetszőleges műveletet jelölhet.

```
x = 5; x += 2; x == 7;
x = 5; x -= 2; x == 3;
x = 5; x *= 2; x == 10;
x = 5; x /= 2; x == 2;
x = 5; x %= 2; x == 1;
```

- A teljes művelet mint kifejezés eredménye a kiszámolt új érték lesz.

A C értékadó műveletei

- Ha e_1 és e_2 kifejezések, akkor

```
e1 ⊙= e2
```

jelentése

```
(e1) = (e1) ⊙ (e2)
```

- Vagyis először külön-külön kiértékelődik az összetett értékadó művelet jobb- és baloldala, majd végrehajtjuk az előírt műveletet, végül pedig az értékadást.
- Az értékadó műveletek jobb-asszociatívak.

C műveletek prioritása

műveletek	asszoc.	C operátorok
egyoperandusú postfix, elemkiválasztás, mezőkiválasztás, függvényhívás	→	`x++`, `x--`, `[]`, `.`, `->`, `f()`
egyoperandusú prefix, méret, típuskényszerítés	←	`+`, `-`, `++x`, `--x`, `!`, `~`, `&`, `*`, `sizeof`, `(type)`
multiplikatív	→	`*`, `/`, `%`
additív	→	`+`, `-`
bitléptetés	→	`<<`, `>>`
kisebb-nagyobb relációs	→	`<=`, `>=`, `<`, `>`
egyenlő-nem egyenlő relációs	→	`==`, `!=`
bitenkénti 'és'	→	`&`
bitenkénti 'kizáró vagy'	→	`^`
bitenkénti 'vagy'	→	`|`
logikai 'és'	→	`&&`
logikai 'vagy'	→	`||`
feltételes	←	`?:`
értékadó	←	`=`, `+=`, `-=`, `*=`, `/=`, `%=` `>>=`, `<<=`, `&=`, `^=`, `|=`
szekvencia	→	`,`

Értékadó műveletek kifejezésekben

- Óvatosan és csak a céljának megfelelően szabad használni ezeket a műveleteket!
- Mi lesz a kiírt érték?

```c
int i = 7;
printf("%d\n", i-- * i++);
```

  - A C szabvány szerint az aritmetikai részkifejezések kiértékelésének sorrendje tetszőleges.
  - `[i==7] i--*i++` → `[i==6] 7*i++` → `[i==7] 7*6` → 42?
  - `[i==7] i--*i++` → `[i==8] i--*7` → `[i==7] 8*7` → 56?
  - Nos, egyes gcc verziókban többféle optimalizálás után még a 49-es eredmény is előfordul.
- Tehát `i` értékét csak akkor inkrementáljuk vagy dekrementáljuk, ha `i` sehol máshol nem szerepel az egész kifejezésben.

Számsorozat jellemzői [1/2]

minimax.c [1–22]

```c
/* Határozzuk meg egy valós számsorozat legkisebb
 * és legnagyobb elemét, valamint a sorozat átlagát!
 * 1997. Október 25.  Dévényi Károly, devenyi@inf.u-szeged.hu
 */

#include <stdio.h>

int main() {
    double vegjel, szam, osszeg, min, max, atlag;
    int db;                                  /* az összegzett elemek száma */

    printf("Ez a program valós számsorozat minimális,\n");
    printf("maximális elemét és átlagát számolja.\n");
    printf("Az input sorozatot végjel zárja.\n");
    printf("Kérem a végjelet!\n");                          /* inicializálás */
    scanf("%lf", &vegjel);
    printf("Kérem az input számsorozatot!\n");
    printf("? ");
    scanf("%lf", &szam);
    min = max = szam;
    osszeg = 0.0;
    db = 0;
```

Számsorozat jellemzői [2/2]

minimax.c [24–46]

```
24      while (szam != vegjel) {                                /* a ciklus kezdete */
25          osszeg += szam;                                         /* összegzés */
26          ++db;                                               /* számláló növelés */
27
28          if (szam < min) {                                   /* min-max számítás */
29              min = szam;
30          } else if (szam > max) {
31              max = szam;
32          }
33                                                  /* a következő szám beolvasása */
34          printf("? ");
35          scanf("%lf", &szam);
36      }                                                           /* a ciklus vége */
37
38      if (db == 0) {
39          printf("Üres számsorozat érkezett.\n");
40      } else {
41          atlag = osszeg / db;
42          printf("Minimum  = %10.3lf Maximum = %10.3lf\n", min, max);
43          printf("Az átlag = %10.3lf\n", atlag);
44      }
45      return 0;
46  }
```

Szinusz(x) kiszámítása

Problémafelvetés és specifikáció

- Problémafelvetés
  - Egy processzorgyártó cég az elhíresült *Pentium FDIV bug* óta különös figyelmet szentel a processzorok működésének átfogó tesztelésére. Jelenleg éppen a trigonometrikus számításokat ellenőrzik. Ezt úgy próbálják megtenni, hogy a szinusz(x) értékét az alapvető műveletekkel kiszámoltatják (adott pontosságon), és ezt majd összehasonlítják a dedikáltan ilyen számításokat végző műveletek eredményével.
  - Ehhez kellene egy olyan program, ami képes kiszámolni a szinusz(x) közelítő értékét a 4 matematikai alapművelet segítségével, a processzorba épített trigonometrikus képességek használata nélkül.
- Specifikáció
  - A probléma inputja egy x valós szám.
  - Az output a $\sin(x)$ közelítő értéke.
  - Az eredményt 10^{-10} pontossággal kell kiszámítani.
  - A szinusz kiszámításához nem használhatjuk a beépített trigonometrikus függvényeket.

Szinusz(x) kiszámítása

Algoritmustervezés 1.

- Ismeretes, hogy

$$\sin(x) = \sum_{i=0}^{\infty} -1^i \frac{x^{(2i+1)}}{(2i+1)!}$$

  A $\sin(x)$ értéke közelíthető ezen sor egy véges kezdőszeletével.
- Akármekkora legyen is a sorfejtés következő tagja, az $\frac{x-1}{2}$-ik tag után a további tagok értéke monoton csökkenő lesz. Mivel alternáló összegről van szó, ha a következő tag értéke kisebb az elvárt pontosságnál, biztosak lehetünk benne, hogy a $\sin(x)$ értékét legalább ilyen pontossággal megközelítettük, tehát befejezhetjük a számolást.

Szinusz(x) kiszámítása

Algoritmustervezés 2.

- Nyilvánvalóan nem célszerű az összes tagot egyedileg kiszámolni, hiszen a következő tag viszonylag egyszerűen számolható az előzőből, és még időt is spórolunk az előző érték felhasználásával.
- Ráadásul a tag számlálójának és nevezőjének külön számolása még pontatlanná is tenné a számítást, mert mindkettő gyorsan növekszik i függvényében.
- A valós típus pontatlansága miatt még így is figyelni kell arra, hogy ne legyen túl nagy eltérés a számolás egyes részeredményei között. Ezt elérhetjük úgy, hogy kihasználva a szinusz periodikusságát, a lehető legkisebb abszolút értékű x-re számoljuk ki a $\sin(x)$ értékét úgy, hogy az eredeti érték szinuszával megegyező eredményt kapjuk.
- Itt is megjegyeznénk, hogy az *összegzés* programozási tételét alkalmazzuk, több szinten is.

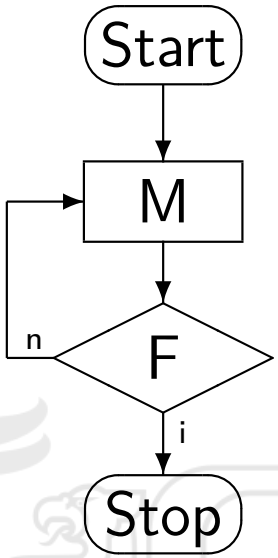
Végfeltételes ismétléses vezérlés

- Végfeltételes vezérlésről akkor beszélünk, ha a ciklusmag elhagyását egy kilépési feltételhez kötjük.
- Legyen F logikai kifejezés, M pedig tetszőleges művelet. Az F kilépési feltételből és az M műveletből (a ciklusmagból) képzett végfeltételes ismétléses vezérlés a következő vezérlési előírást jelenti:
  1. Hajtsuk végre az M műveletet majd folytassuk a 2.) lépéssel.
  2. Értékeljük ki az F feltételt és folytassuk a 3.) lépéssel.
  3. Ha F értéke igaz, akkor az ismétléses vezérlés és ezzel együtt az összetett művelet végrehajtása befejeződött.
  4. Egyébként, vagyis ha az F értéke hamis, akkor folytassuk az 1.) lépéssel.

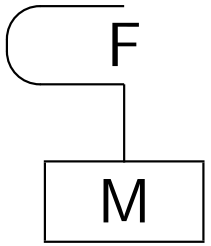
Végfeltételes ismétléses vezérlés

Folyamatábra és szerkezeti ábra

- Folyamatábrája:



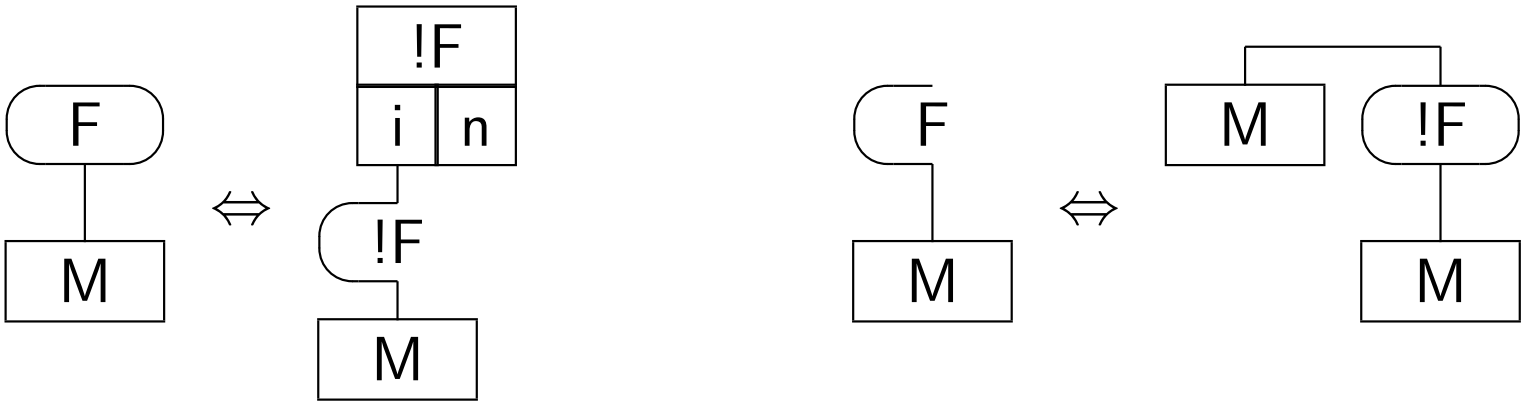
- Szerkezeti ábrája:



- Ha az F értéke kezdetben hamis, és az M művelet nincs hatással F-re, akkor végtelen ciklust kapunk.
- Ha az F értéke kezdetben igaz, M legalább egyszer akkor is végrehajtásra kerül.

Kezdő- és végfeltételes vezérlések kapcsolata

- A kezdő és végfeltételes vezérlések kifejezhetőek egymás segítségével.

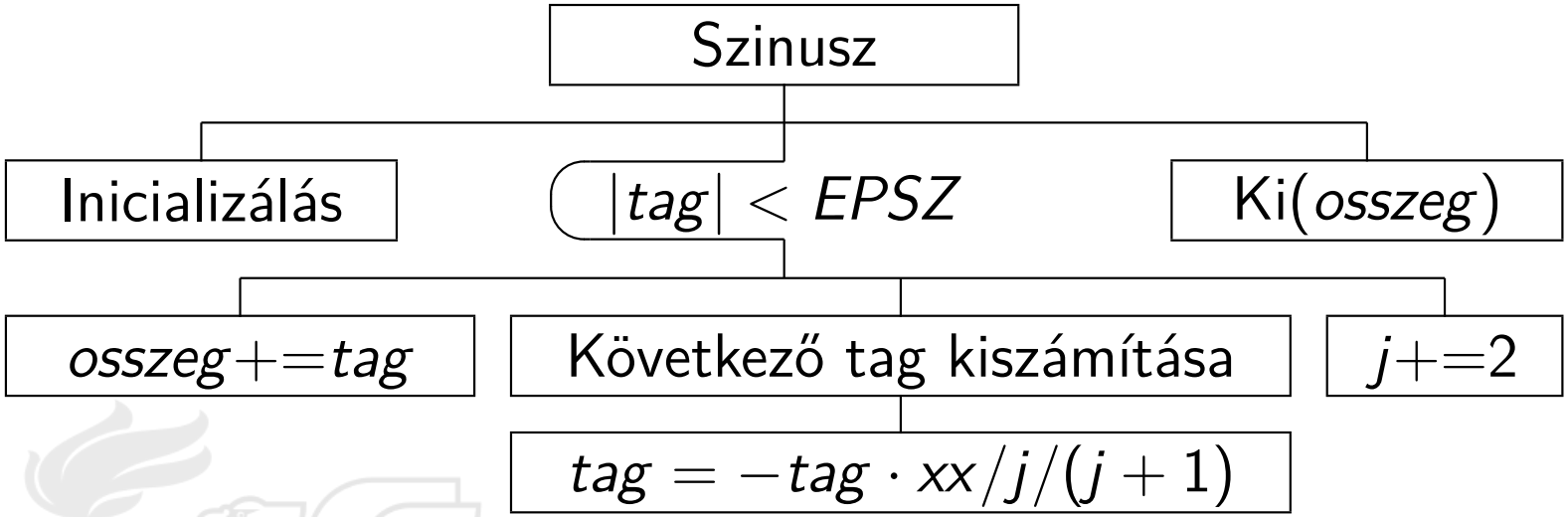


- Az algoritmus tervezésekor előfordulhat, hogy a kezdő- és végfeltételes vezérlés is alkalmasnak látszik a probléma megoldására. Ilyenkor érdemes megvizsgálni, hogy az *F* feltétel szükséges feltétele-e az *M* művelet végrehajtásának? Ha igen, akkor kezdőfeltételes ismétléses vezérlést kell választani.

Szinusz(x) kiszámítása

Algoritmustervezés – Szerkezeti ábra 1.

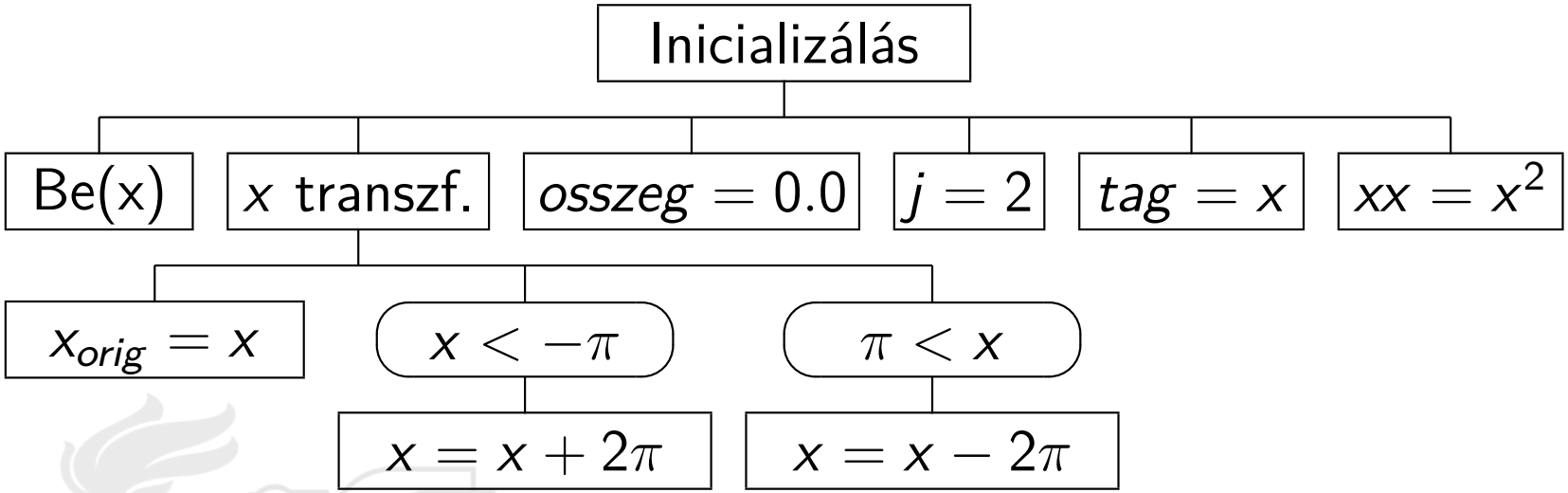
- A főprogram:



Szinusz(x) kiszámítása

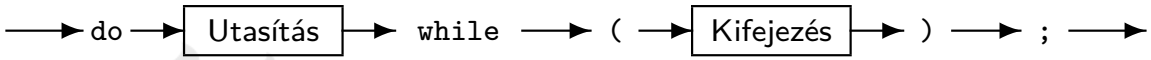
Algoritmustervezés – Szerkezeti ábra 2.

- Inicializálás:



A `do while` utasítás

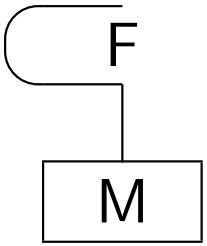
- Ha valamilyen műveletet mindaddig végre kell hajtani, amíg egy feltétel igaz a `do while` utasítás (is) használható.
  - A művelet végrehajtása szükséges a feltétel kiértékeléséhez.
  - A feltétel ellenőrzése a művelet után történik, így ha a feltétel kezdetben hamis volt, a műveletet akkor is legalább egyszer végrehajtjuk.
- A `do while` utasítás szintaxisa C-ben



Végfeltételes ismétléses vezérlés

Megvalósítás C nyelven

- Szerkezeti ábrája:



- Megvalósítása:

```
do {
     M;
} while (!F);
```

Szinusz(x) kiszámítása [1/2]

szinusz.c [1–22]

```c
/* sin(x) közelítő értékének kiszámítása a beépített sin(x)
 *    függvény alkalmazása nélkül.
 *    x értékét transzformáljuk a (-Pi,Pi) intervallumba.
 * 1997. Október 25.  Dévényi Károly, devenyi@inf.u-szeged.hu
 */

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

#define EPSZ    1e-10                          /* a közelítés pontossága */

int main() {
    double osszeg;                           /* a végtelen sor kezdőösszege */
    double tag;                             /* a végtelen sor aktuális tagja */
    double x;                                                 /* argumentum */
    double x_orig;                        /* az argumentum értékének megőrzése*/
    double xx;                                                  /* sqr(x) */
    int j;                                      /* a nevező kiszámításához */

    printf("Kérem␣sin(x)-hez␣az␣argumentumot\n");
    scanf("%lg%*[^\n]", &x); getchar();              /* sor végéig olvasunk */
    x_orig = x;
```

Szinusz(x) kiszámítása [2/2]

szinusz.c [24–44]

```
24      while (x < -M_PI) {                                     /* transzformálás */
25          x += 2 * M_PI;
26      }
27      while (M_PI < x) {
28          x -= 2 * M_PI;
29      }
30
31      osszeg = 0.0;
32      j      = 2;                                              /* inicializálás */
33      tag    = x;
34      xx     = x * x;
35
36      do {                                                    /* ciklus kezdete */
37          osszeg += tag;
38          tag = -(tag * xx / j / (j + 1));                    /* következő tag */
39          j += 2;
40      } while (fabs(tag) >= EPSZ);                /* végfeltétel, ciklus vége */
41
42      printf("sin(%8.5lf)= %13.10lf\n", x_orig, osszeg);
43      return 0;
44  }
```

Tartalom

Eljárásvezérlés

- Eljárásvezérlésről akkor beszélünk, amikor egy műveletet adott argumentumokra alkalmazunk, aminek hatására az argumentumok értékei pontosan meghatározott módon változnak meg.
- Az eljárásvezérlés fajtái:
  - Eljárásművelet
  - Függvényművelet

Eljárásművelet

- Eljárásműveleten olyan tevékenységet értünk, amelynek alkalmazása adott argumentumokra az argumentumok értékének pontosan meghatározott megváltozását eredményezi.
- Minden eljárásműveletnek rögzített számú paramétere van, és minden paraméter rögzített adattípusú.
- Minden paraméter rögzített kezelési módú, ami lehet:
  - Bemenő mód
    - Ha a művelet végrehajtása nem változtathatja meg az adott argumentum értékét.
  - Kimenő mód
    - Ha a művelet eredménye nem függ az adott argumentum végrehajtás előtti értékétől, de az adott argumentum értéke a művelet hatására megváltozhat.
  - Be- és kimenő (vegyes) mód
    - Ha a művelet felhasználhatja az adott argumentum végrehajtás előtti értékét és az argumentum értéke a művelet hatására meg is változhat.

Függvényművelet

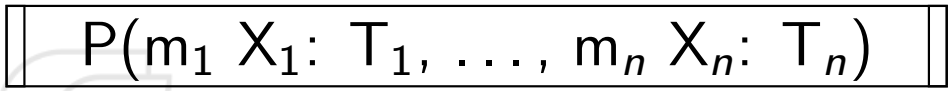
- A matematikai függvény fogalmának általánosítása.
- Ha egy részprobléma célja egy érték kiszámítása adott értékek függvényében, akkor a megoldást megfogalmazhatjuk függvényművelettel.
- A függvényművelet argumentumai ugyanúgy lehetnek kimenő és be- és kimenő módúak is, mint az eljárásműveletek esetén, tehát a függvényművelet végrehajtása az aktuális argumentumok megváltozását is eredményezheti.

Eljárás és függvény specifikációja

- Az eljárásművelet specifikációja tartalmazza:
  - A művelet elnevezését
  - A paraméterek felsorolását
  - Mindegyik paraméter adattípusát
  - A művelet hatásának leírását
- A függvényművelet specifikációja a fentieken túl tartalmazza még:
  - A függvényművelet eredménytípusát

Eljárásművelet általános jelölése

- Eljárásműveletet $\texttt{P(m}_1\ \texttt{X}_1\ \texttt{:}\ \ \texttt{T}_1\texttt{;}\ \ldots\texttt{;}\ \texttt{m}_n\ \texttt{X}_n\ \texttt{:}\ \ \texttt{T}_n\texttt{)}$ formában jelölhetünk, ahol
  - $\texttt{P}$ az eljárásművelet neve
  - $\texttt{m}_i$ az i. paraméter kezelési módja
  - $\texttt{X}_i$ az i. paraméter azonosítója
  - $\texttt{T}_i$ az i. paraméter adattípusa
- Az eljárás algoritmusa egy olyan szerkezeti ábrával adható meg, melynek a feje így néz ki:



Eljáráshívás

- A fent jelölt eljárásművelet adott $A_1, \ldots, A_n$ argumentumokra történő végrehajtását eljáráshívásnak nevezzük és a `P(`A_1`, ..., `A_n`)` jelölést használjuk.
- Ha az i. paraméter kezelési módja kimenő vagy be- és kimenő, akkor az A_i aktuális argumentum csak változó lehet.
- Az eljáráshívás utasítás.

Függvényműveletek általános jelölése

- Függvényművelet `F(`$\mathtt{m}_1$ $\mathtt{X}_1$ `:` $\mathtt{T}_1$`; ...;` $\mathtt{m}_n$ $\mathtt{X}_n$ `:` $\mathtt{T}_n$`) :` `T` formában jelölhetünk, ahol
  - `F` a függvényművelet neve
  - $\mathtt{m}_i$ az i. paraméter kezelési módja
  - $\mathtt{X}_i$ az i. paraméter azonosítója
  - $\mathtt{T}_i$ az i. paraméter adattípusa
  - `T` a függvényművelet eredménytípusa
- A függvény algoritmusa egy olyan szerkezeti ábrával adható meg, melynek a feje így néz ki:

$$\boxed{\mathtt{F}(\mathtt{m}_1\ \mathtt{X}_1\colon\ \mathtt{T}_1,\ \ldots,\ \mathtt{m}_n\ \mathtt{X}_n\colon\ \mathtt{T}_n)\colon\ \mathtt{T}}$$

- Továbbá a szerkezeti ábrában lennie kell (legalább) egy olyan utasításnak, amely a függvény aktuális argumentumokra számított értékével visszatér a függvényből.

Függvényhívás

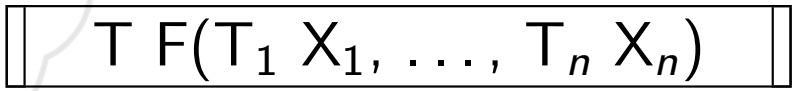
- A fent jelölt függvényműveletnek adott $A_1, \ldots, A_n$ argumentumokra történő végrehajtását függvényhívásnak nevezzük és az `F(`A_1`, ..., `A_n`)` jelölést használjuk.
- Ha az i. paraméter kezelési módja kimenő vagy be- és kimenő, akkor az A_i aktuális argumentum csak változó lehet.
- A függvényhívás kifejezés.

Eljárásvezérlés megvalósítása

- Vannak olyan programozási nyelvek, ahol a függvény és eljárásműveletek meg vannak különböztetve, valamint a paraméterek módjaira sincs megkötés.
- Mivel azonban e kurzus keretében csak a C nyelvről lesz szó, a szerkezeti ábrán a továbbiakban (legalábbis a legtöbb esetben) igazodni fogunk a C nyelvhez.
  - A C nyelvben lényegében csak függvényművelet van.
  - C nyelvben a függvényművelet argumentumai bemenő módúak, tehát alapvetően a függvényművelet végrehajtása az aktuális argumentumok megváltozását nem eredményezheti.

Függvényművelet

- A függvényművelet jelölésére a `T F(T1 X1, ..., Tn Xn)` formát használjuk, ahol
  - `T` a függvényművelet eredménytípusa
  - `F` a függvényművelet neve
  - $\texttt{T}_i$ az i. paraméter adattípusa
  - $\texttt{X}_i$ az i. paraméter azonosítója
- A zárójeleket üres paraméterlista esetén is ki kell tenni.
- A C jelölésmódhoz igazodva, a függvény algoritmusa egy olyan szerkezeti ábrával adható meg, melynek a feje így néz ki:



- Továbbá a szerkezeti ábrában lennie kell (legalább) egy olyan `return` utasításnak, amely visszaadja a függvény által kiszámított értéket.

Függvényművelet

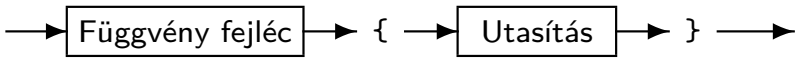
- A fent jelölt függvényműveletnek adott $A_1, \ldots, A_n$ aktuális argumentumokra történő végrehajtását függvényhívásnak nevezzük és az `F(`A_1`, ..., `A_n`)` jelölést használjuk.
- A függvényhívás kifejezés.
- A zárójeleket paraméter nélküli függvény hívása esetén is ki kell tenni.

Függvény

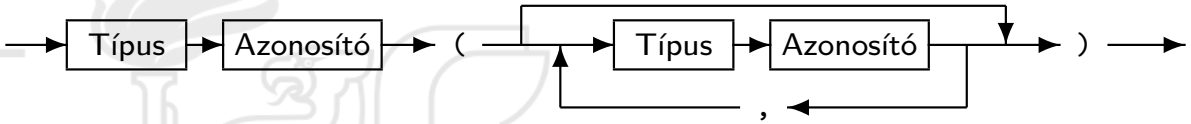
Szintaxis

- Függvény deklaráció

- Függvény definíció (egyben deklaráció is)



- Függvény fejléc



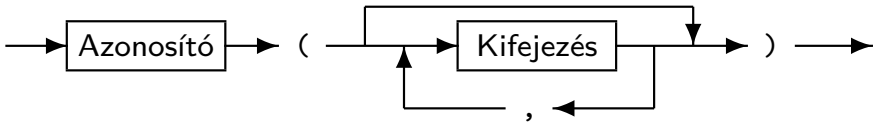
A `return` utasítás

- Minden függvényben szerepelnie kell legalább egy `return` utasításnak.
- Ha a függvényben egy ilyen utasítást hajtunk végre, akkor a függvény értékének kiszámítása befejeződik. A hívás helyén a függvény a `return` által kiszámított értéket veszi fel.
- A `return` utasítás szintaxisa C-ben

Függvényhívás

Szintaxis

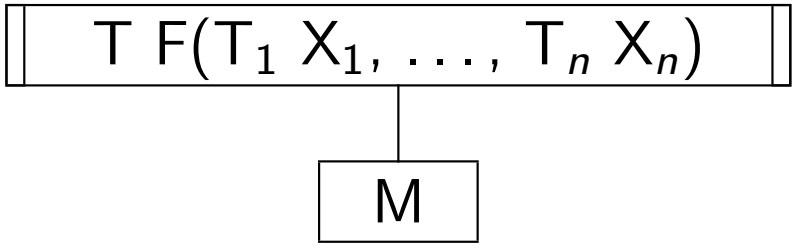
- Függvényhívás szintaxisa C-ben



Függvényművelet

Megvalósítás C nyelven

- Szerkezeti ábrája:



- Megvalósítása:

```
T F(T1 X1, ..., Tn Xn)
{
    M;
}
```

Függvényművelet

Példa

- A szabvány szerinti deklaráció:

```
double atlag(double a, double b) {
    return (a + b) / 2;
}
```

- Régen C-ben így kellett függvényt deklarálni:

```
double atlag(a,b)
    double a, double b;
{
    return (a + b) / 2;
}
```

Eljárásművelet

Megvalósítás C nyelven

- Ha eljárást szeretnénk készíteni C nyelven, akkor egy olyan függvényt kell deklarálni, melynek eredménytípusa `void`. Ebben az esetben a függvény definíciójában nem kötelező a `return` utasítás, illetve ha mégis van ilyen, akkor nem adható meg utána kifejezés.

Vegyes és kimenő módú argumentumok

Megvalósítás C nyelven

- Mint említettük, C-ben csak bemenő módú argumentumok vannak. De mi magunk kezelhetjük a be- és kimenő illetve kimenő módú argumentumokat pointerek segítségével.
- Az alábbiakban egy működő megoldást mutatunk, egyelőre részletes magyarázat nélkül:
  - Ha az i. paramétert kimenő (vagy vegyes) módúnak szeretnénk, akkor a függvény deklarációjában `T`$_i$ `X`$_i$ helyett `T`$_i$ `*X`$_i$ deklarációt, a függvénytörzsben pedig `X`$_i$ helyett mindenhol `(*X`$_i$`)` változóhivatkozást használunk.
  - Továbbá a függvény meghívásakor az `A`$_i$ argumentum helyett az `&A`$_i$ argumentumot használjuk.
- Részletesebb magyarázatot a pointerek megismerése után adunk.

Kamatos kamat számítása

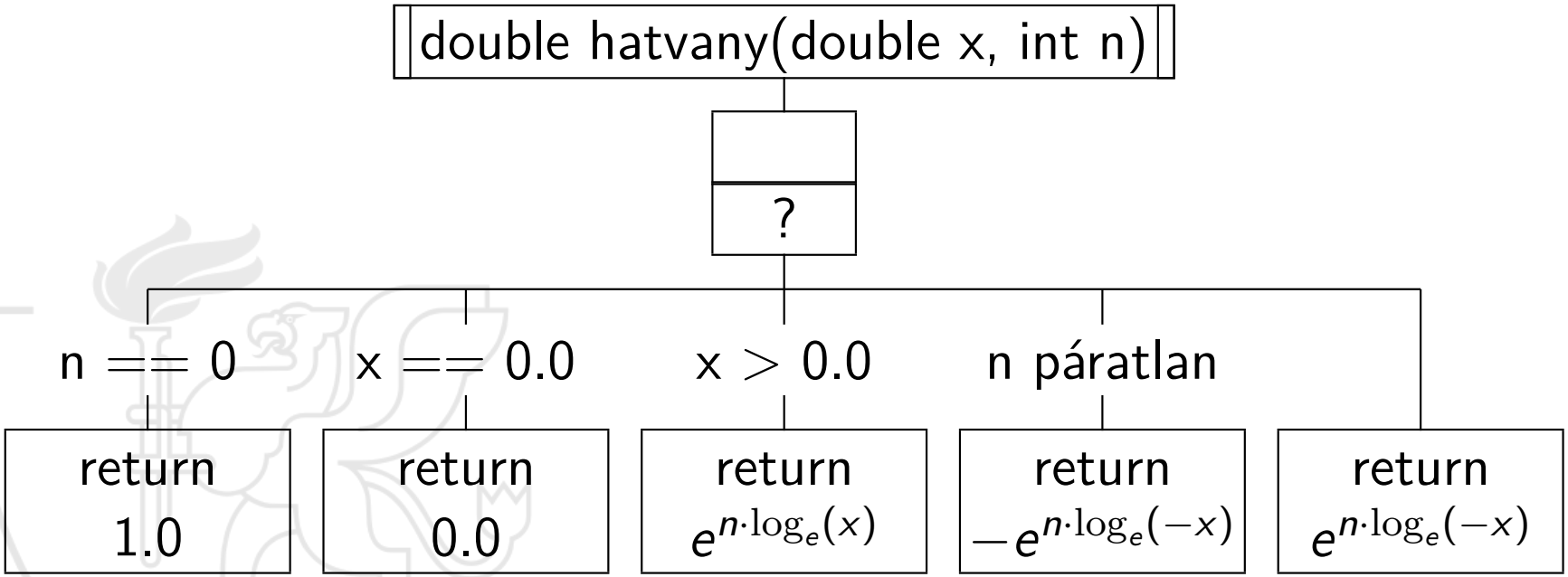
Problémafelvetés és specifikáció

- Problémafelvetés
  - A szakemberek régóta figyelmeztetnek, hogy a jelenlegi nyugdíjrendszer a jelenlegi körülmények között hosszú távon nem fentartható, ezért mindenki gondoskodjon magáról. Ennek egyik módja lehet, ha valaki egy nagyobb összeget hosszabb időre leköt. Ennek a megtakarításnak a jellemzője, hogy az éves kamat minden év végén hozzáadódik a tőkéhez, és együtt kamatoznak tovább. Szeretnénk tudni, hogy ha ilyen módon lekötünk egy összeget fix éves kamatszint mellett több évre, akkor a végén mennyi pénzt tudunk majd felvenni.
  - Kellene egy program, ami kamatos kamatot tud számolni!
- Specifikáció
  - A probléma inputja a betett összeg (valós), az éves kamatláb (egész, százalékos érték) és az évek száma (egész).
  - Az adott időre betett összeg éves alapon tőkenövelt kamatozással adott kamatlábbal számított új értéke a futamidő végén.

Kamatos kamat számítása

Algoritmustervezés – Szerkezeti ábra

- A kamatos-kamat számításhoz készítünk egy általános hatványozó függvényt és ezt hívjuk meg a kamatos-kamat számítás alapképletével.
- Az általános hatványozó függvény szerkezeti ábrája:



Bitenkénti logikai műveletek

- Mint említettük, a C nyelv a magas szintű struktúrái mellett (pl. vezérlési szerkezetek) az alacsony szintű programozást is nyelvi szinten elérhetővé teszi.
- Ennek szellemében a C nyelv egész típusú (`int` és `char`) értékeire definiálva vannak a bitmanipulációs operátorok.
  - Ezek a műveletek az egész értéket annak bináris számábrázolási (a típus által meghatározott méretű kettes számrendszerbeli vagy kettes komplemens) alakjával megegyező bitsorozatként kezelik.
  - Logikai értékként a 0 értékű bit a hamis, az 1 értékű bit az igaz értéket jelenti.

Az `int` (egész) adattípus műveletei

Bitenkénti logikai műveletek

- `int`→`int`
  (*egész*→*egész*)
  egy operandusú műveletek
  - `~` bitenkénti negáció
- `int`×`int`→`int`
  (*egész*×*egész*→*egész*)
  két operandusú műveletek
  - `&` bitenkénti *és*
  - `|` bitenkénti *vagy*
  - `^` bitenkénti *kizáró vagy*
  - `<<` balra léptetés
  - `>>` jobbra léptetés

```
~a
a & 1
a | 0xf0
a ^ b
1 << a
a >> 3
```

C műveletek prioritása

műveletek	asszoc.	C operátorok
egyoperandusú postfix, elemkiválasztás, mezőkiválasztás, függvényhívás	→	`x++`, `x--`, `[]`, `.`, `->`, `f()`
egyoperandusú prefix, méret, típuskényszerítés	←	`+`, `-`, `++x`, `--x`, `!`, `~`, `&`, `*`, `sizeof`, `(type)`
multiplikatív	→	`*`, `/`, `%`
additív	→	`+`, `-`
bitléptetés	→	`<<`, `>>`
kisebb-nagyobb relációs	→	`<=`, `>=`, `<`, `>`
egyenlő-nem egyenlő relációs	→	`==`, `!=`
bitenkénti 'és'	→	`&`
bitenkénti 'kizáró vagy'	→	`^`
bitenkénti 'vagy'	→	`|`
logikai 'és'	→	`&&`
logikai 'vagy'	→	`||`
feltételes	←	`?:`
értékadó	←	`=`, `+=`, `-=`, `*=`, `/=`, `%=` `>>=`, `<<=`, `&=`, `^=`, `|=`
szekvencia	→	`,`

Kamatos kamat számítása

Megvalósítás

- Egy `X` szám páratlanságát többféleképpen ellenőrizhetjük:
  - A kettővel való osztás maradéka 1:

```
(X % 2) == 1
```

  - A szám kettes számrendszerbeli alakjának utolsó számjegye 1. Ehhez bitműveletet használunk: a szám összes többi számjegyét töröljük, vagyis a legalsó helyiértékű bit kivételével mindet 0-ra állítjuk:

```
(X & 1) == 1
```

  - A bitműveletekről később részletesen is lesz szó.

Kamatos kamat számítása [1/2]

kamatos.c [1–22]

```c
/* Kamatos-kamat számítás a hatványozás függvény segítségével.
 * 1997. Október 31.  Dévényi Károly, devenyi@inf.u-szeged.hu
 * 2014. Szeptember 23. Gergely Tamás, gertom@inf.u-szeged.hu
 */

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

double hatvany(double x, int n) {
                                    /* x n-edik hatványát kiszámító függvény */
    if (n == 0) {
        return 1.0;
    } else if (x == 0.0) {
        return 0.0;
    } else if (x > 0.0) {
        return exp(n * log(x));
    } else if (n & 1) {
        return -exp(n * log(-x));
    } else {
        return exp(n * log(-x));
    }
}
```

Kamatos kamat számítása [2/2]

kamatos.c [24–41]

```
24 int main() {
25     double osszeg, uj_osszeg;
26     int kamatlab, ev;
27 
28     printf("␣A␣kamatozó␣összeg␣?␣");
29     scanf("%lg%*[^\n]", &osszeg);
30     getchar();
31     printf("␣A␣kamatláb␣?␣");
32     scanf("%d%*[^\n]", &kamatlab);
33     getchar();
34     printf("␣A␣kamatozási␣évek␣száma␣?␣");
35     scanf("%d%*[^\n]", &ev);
36     getchar();
37     uj_osszeg = osszeg * hatvany(1.0 + kamatlab / 100.0, ev);
38     printf("A␣kamatos␣kamattal␣növelt␣összeg:");
39     printf("%10.2lf\n", uj_osszeg);
40     return 0;
41 }
```

Blokkstruktúra a C nyelvben

- A C nyelvben blokkon egy { és } zárójelpárba zárt programrészt értünk.
- Egy C program blokkjai mellérendeltségi és alárendeltségi viszonyban vannak. Ezt a viszonyt az ide vonatkozó szabályokkal együtt blokkstruktúrának nevezzük.
- A blokkstruktúra az azonosítók *láthatóságát* befolyásolja, de hatással lehet például a változók *létezésére* is.

Blokkstruktúra a C nyelvben

Szabályok

- Sorrendiségi szabály
  - A program egy adott pontján csak a hivatkozás helyét megelőzően már deklarált azonosítóra hivatkozhatunk. Változó-, függvény- és típus-azonosító a megjelenése helyén deklaráltnak minősül.
- Egyediségi szabály
  - Egy adott blokkban egy azonosító csak egyszer deklarálható, nem számítva az alárendelt blokkokat.
- Láthatósági szabály
  - Egy B_1 blokkban deklarált A azonosító akkor és csak akkor látható (hivatkozható) egy B_2 blokkban, ha
    - B_1 megegyezik B_2-vel, vagy B_2 alárendeltje B_1-nek és az A azonosító előbb van deklarálva, mint B_2, és
    - az A azonosító nincs deklarálva egyetlen olyan C blokkban sem, amely alárendeltje B_1-nek és amelynek B_2 alárendeltje (beleértve azt, hogy B_2 megegyezik C-vel).

Blokkstruktúra a C nyelvben

Szabályok

```
 int main() {
►    int b, b;
     a = 0;
     printf("? ");
►    scanf("%d", &b);
►    b = f(b);
►    printf("%d (%d)\n", b, a);
 }
 int a;
 int f(int n) {
     a++;
     return (n>1) ?
         f(n-1) + f(n-2) : 1;
 }
```

- A ► ponton egyszerre két változót deklarálunk, és mindkettőnek a b nevet adjuk. Ha ezt megtehetnénk, hogy döntené el a fordító a ► pontokon, hogy melyik esetben melyik változóval kellene dolgoznia?

Blokkstruktúra a C nyelvben

Szabályok

```
  int main() {
      int b, b;
►     a = 0;
      printf("? ");
      scanf("%d", &b);
►     b = f(b);
►     printf("%d (%d)\n", b, a);
  }
►int a;
►int f(int n) {
      a++;
      return (n>1) ?
          f(n-1) + f(n-2) : 1;
  }
```

- Itt pedig a ► pontokon használjuk az `f` és az `a` azonosítókat, holott még azt sem tudjuk, hogy melyik micsoda, mert az csak a ► pontokon derül ki.

Blokkstruktúra a C nyelvben

Szabályok

```
 int main() {
     int b, b;
     a = 0;
     printf("? ");
     scanf("%d", &b);
     b = f(b);
     printf("%d (%d)\n", b, a);
 }
▶int a;
▶int f(int n) {
▶    a++;
     return (n>1) ?
▶        f(n-1) + f(n-2) : 1;
 }
```

- Itt a ▶ pontokon már jogos az `f` és az `a` használata is, mert a ▶ pontokon már deklarálva lettek, még ha `f` definíciója nincs is teljesen befejezve.

Blokkstruktúra a C nyelvben

Szabályok

```
 {                    /* 1. BLOKK */
►    int a, b, c;
     {                /* 2. BLOKK */
          float b;
►         a = 2;
     }
     b = 1;
     {                /* 3. BLOKK */
          float c;
          {           /* 4. BLOKK */
               c = 3.4;
          }
     }
 }
```

- A ► ponton használható a ► ponton deklarált a változó mert:
  - a 2. blokk alárendeltje az 1. blokknak,
  - a előbb van deklarálva mint a 2. blokk,
  - nincs olyan blokk a kettő között, amelyben a deklarálva lenne.

Blokkstruktúra a C nyelvben

Szabályok

```
 {                    /* 1. BLOKK */
►    int a, b, c;
     {                /* 2. BLOKK */
►        float b;
         a = 2;
     }
►    b = 1;
     {                /* 3. BLOKK */
         float c;
         {            /* 4. BLOKK */
             c = 3.4;
         }
     }
 }
```

- A ► ponton a ► ponton deklarált b változó használható.
- Igaz ugyan, hogy a ► ponton is deklarálunk egy b változót, de ez csak a 2. blokkban és az alárendelt blokkokban érvényes, de az 1. blokk nem alárendeltje a 2. blokknak.

Blokkstruktúra a C nyelvben

Szabályok

```
 {                    /* 1. BLOKK */
►    int a, b, c;
     {                /* 2. BLOKK */
         float b;
         a = 2;
     }
     b = 1;
     {                /* 3. BLOKK */
►        float c;
         {            /* 4. BLOKK */
►            c = 3.4;
         }
     }
 }
```

- A ► ponton a ► ponton deklarált c változó használható.
- A ► ponton deklarált c változó nem használható, mert a 3. blokk ► pontján deklarált c változó elfedi azt.

Blokkstruktúra a C nyelvben

Hatáskör, globális és lokális azonosítók

- Azon blokkok összességét, amelyből egy *a* azonosító látható, az *a* azonosító hatáskörének nevezzük.
- Egy azonosítót lokálisnak nevezünk egy blokkra nézve, ha az azonosító az adott blokkban van deklarálva.
- Azt mondjuk, hogy egy *a* azonosító globális egy *B* blokkra nézve, ha nem *B*-ben van deklarálva, de látható *B*-ben.
- A blokkstruktúra alapján látható, hogy a C nyelvben vannak úgynevezett lokális változók, sőt általában ezeket használjuk.
- Látható azonban az is, hogy a programfájlban deklarált programegységek globálisak az összes függvénydeklarációra nézve, vagyis ezek minden blokkban láthatóak a deklarálásuktól kezdve az újradeklarálásukig. Ezeket csak nagyon indokolt esetben szoktuk használni.

Blokkstruktúra a C nyelvben

Megjegyzések gcc-hez

- A gcc néha elviseli, ha egy függvényt hamarabb használunk, mint ahogyan deklarálnánk (tehát megsértjük a sorrendiségi szabályt). A hívásból ugyanis ki tudja deríteni a paraméterek számát és típusát, a visszatérési értéket viszont ilyen esetekben `int`-ként kezeli.
- Az ansi C nem engedi meg a deklarációk és utasítások keveredését, tehát már a blokk elején deklarálni kell az összes változót. A C99 szabvány ennél rugalmasabb. A gcc például már warningot is csak a `-pedantic` kapcsolóval ad ilyen esetekre.

Tárolási osztályok

`auto` „Automatikus”: A globálisan deklarált változóknak hely foglalódik a program teljes futási idejére, a blokkokban lokálisan deklarált változóknak pedig az adott blokk végrehajtásának idejére. Ez az alapértelmezett tárolási mód, így az `auto` kulcsszót nem szoktuk kiírni.

`static` „Állandó”: Az ilyen lokálisan deklarált változónak mindenképpen „statikus”, vagyis állandó helyet foglalunk a program teljes futási idejére. A változó értéke megmarad a blokk végrehajtása után is és a blokk újabb végrehajtásakor a megőrzött érték felhasználható. A kulcsszó a lokálisan deklarált változó *létezését* befolyásolja, a *láthatóságát* nem.

`extern` „Külső”: Ezzel jelezzük a fordítónak, hogy a változó létezik, használni fogjuk, de ne foglaljon neki helyet, mert azt valahol máshol tesszük meg. A változóhivatkozások feloldása majd a linker feladata lesz.

A `static` kulcsszó hatása [1/1]

static.c [1–23]

```c
/* A static változót mutatjuk be.
 * 1997. November 7.  Dévényi Károly, devenyi@inf.u-szeged.hu
 */

#include <stdio.h>

void stat();

int main() {
    int i;                                          /* ciklusváltozó */
    for (i = 0; i < 5; ++i) {
        stat();
    }
    return 0;
}

void stat() {
    int ideiglenes = 1;                  /* minden hívásnál inicializálódik */
    static int allando = 1;    /* a program elején egyszer inicializálódik */
    printf("ideiglenes = %d allando = %d\n", ideiglenes, allando);
    ++ideiglenes;
    ++allando;
}
```

Tárolási osztályok

- A példaprogram kimenete

```
$ ./static
ideiglenes = 1 allando = 1
ideiglenes = 1 allando = 2
ideiglenes = 1 allando = 3
ideiglenes = 1 allando = 4
ideiglenes = 1 allando = 5
```

Hanoi tornyai

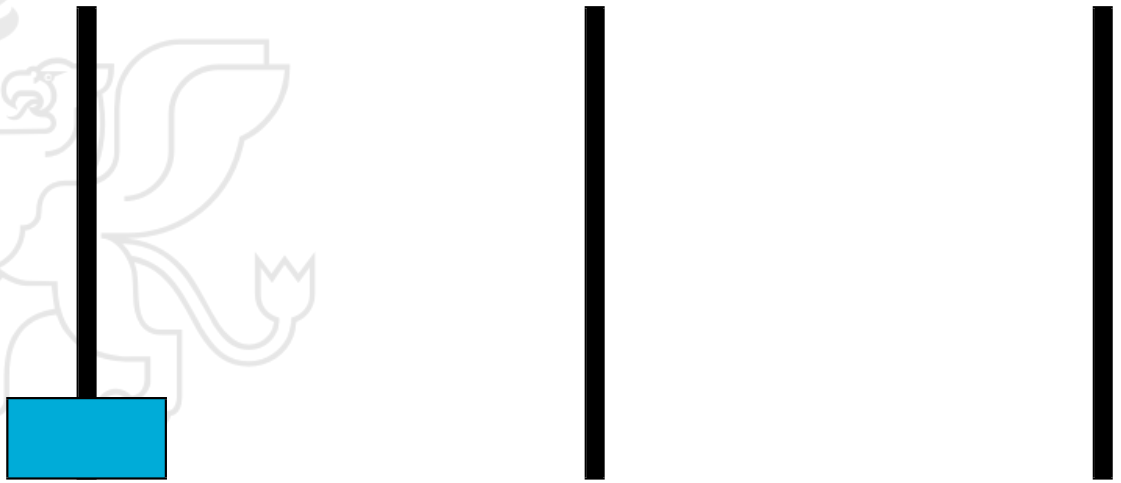
Problémafelvetés és specifikáció

- Problémafelvetés
  - Adjunk megoldást a Hanoi tornyai játékra.
  - A játékot Édouard Lucas, egy francia matematikus találta ki. A játék lényege, hogy egy oszlopon egyre csökkenő átmérőjű korongok vannak, és ezeket a korongokat át kell pakolni egy másik üres oszlopra úgy, hogy
    - egyszerre csak egy korongot mozgathatunk valamelyik oszlop tetejéről egy másik oszlop tetejére,
    - nagyobb átmérőjű korong nem kerülhet kisebb átmérőjű korongra, és
    - rendelkezésre áll egy kezdetben üres harmadik oszlop is.
- Specifikáció
  - Az input a korongok száma (pozitív egész) valamint a kezdő és a cél oszlop sorszáma ($\in \{1, 2, 3\}$).
  - Egy tevékenységsorozat szövegesen, amit mechanikusan végrehajtva szabályosan átpakolhatjuk a tornyot.

Hanoi tornyai

Algoritmustervezés 1.

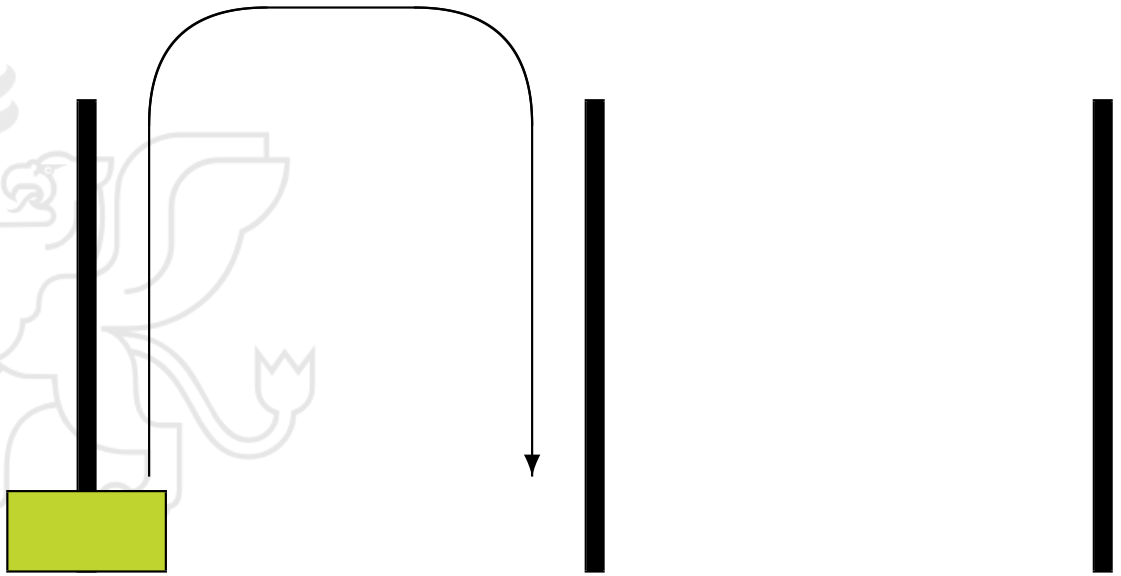
- Készítsünk egy rekurzív eljárást, amelyik az N magasságú torony átpakolását visszavezeti az N-1 magasságú torony átpakolására.
- Az 1 magasságú torony átpakolása nem igényel előkészületet, azonnal elvégezhető.



Hanoi tornyai

Algoritmustervezés 1.

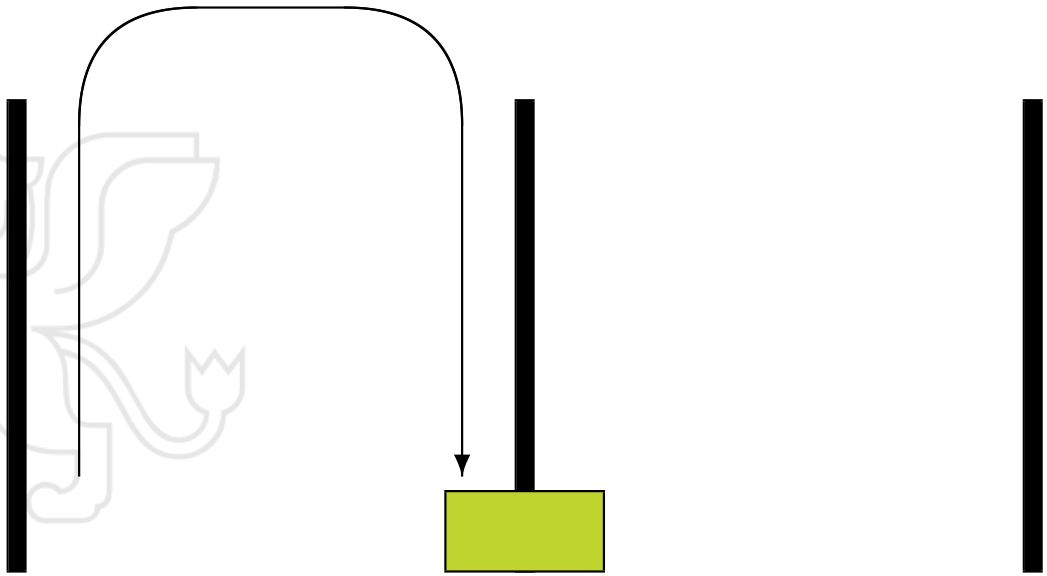
- Készítsünk egy rekurzív eljárást, amelyik az N magasságú torony átpakolását visszavezeti az N-1 magasságú torony átpakolására.
- Az 1 magasságú torony átpakolása nem igényel előkészületet, azonnal elvégezhető.



Hanoi tornyai

Algoritmustervezés 1.

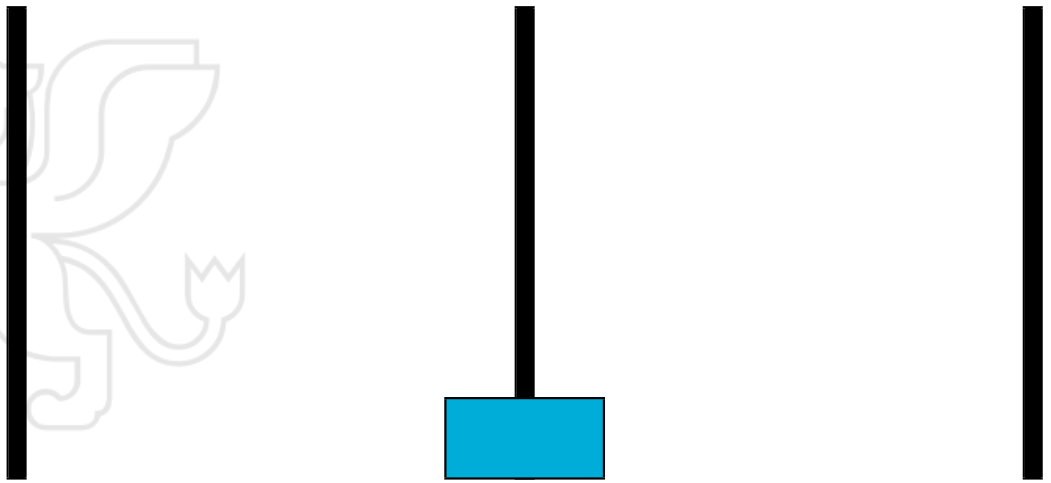
- Készítsünk egy rekurzív eljárást, amelyik az N magasságú torony átpakolását visszavezeti az N-1 magasságú torony átpakolására.
- Az 1 magasságú torony átpakolása nem igényel előkészületet, azonnal elvégezhető.



Hanoi tornyai

Algoritmustervezés 1.

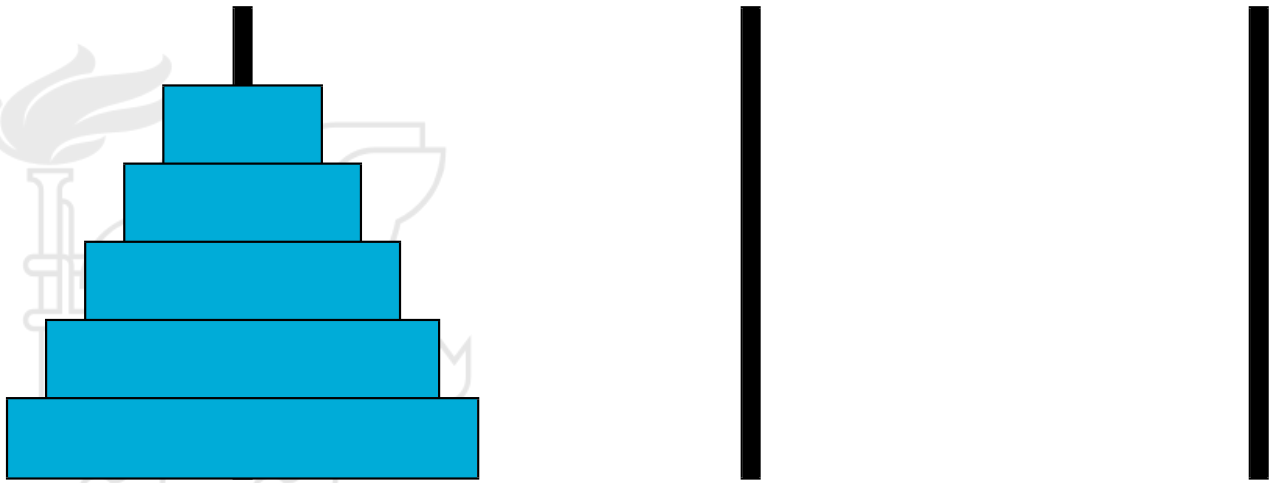
- Készítsünk egy rekurzív eljárást, amelyik az N magasságú torony átpakolását visszavezeti az N-1 magasságú torony átpakolására.
- Az 1 magasságú torony átpakolása nem igényel előkészületet, azonnal elvégezhető.



Hanoi tornyai

Algoritmustervezés 2.

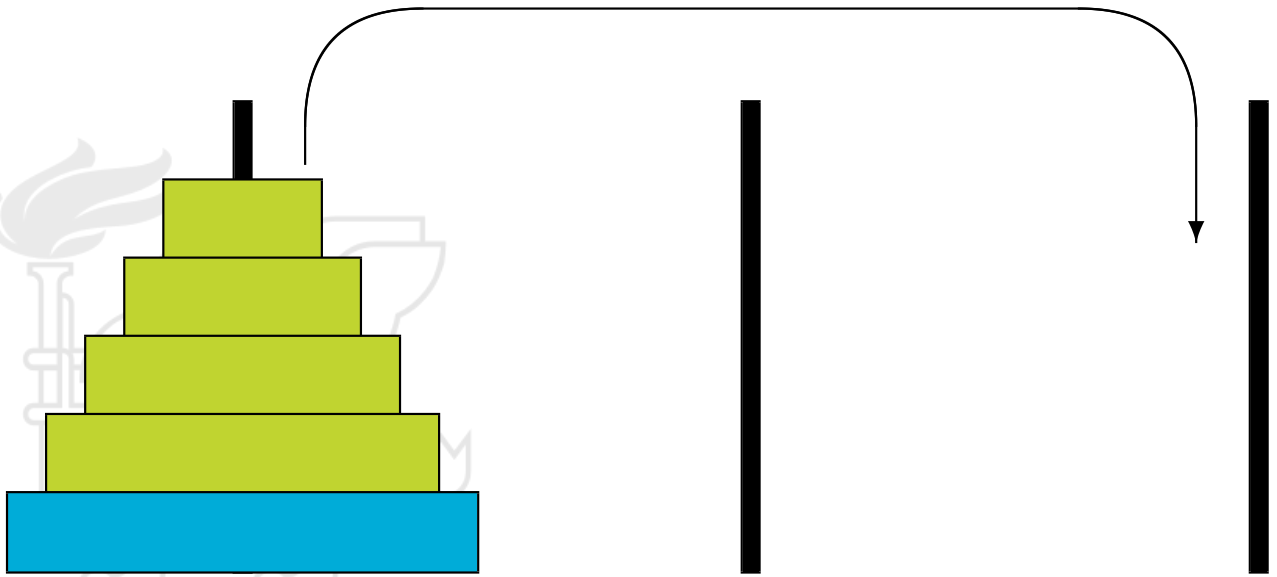
- Az N magasságú torony átpakolását visszavezetjük az N-1 magasságú torony átpakolására.



Hanoi tornyai

Algoritmustervezés 2.

- Az N magasságú torony átpakolását visszavezetjük az N-1 magasságú torony átpakolására.



Hanoi tornyai

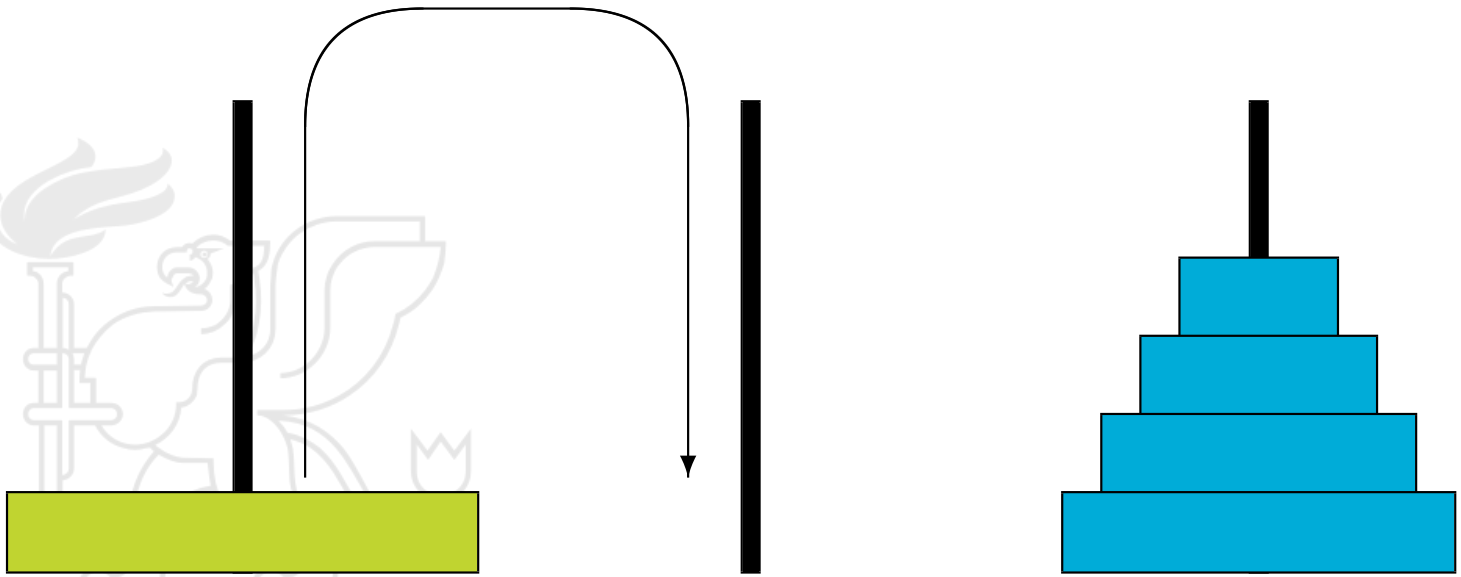
Algoritmustervezés 2.

- Az N magasságú torony átpakolását visszavezetjük az N-1 magasságú torony átpakolására.

Hanoi tornyai

Algoritmustervezés 2.

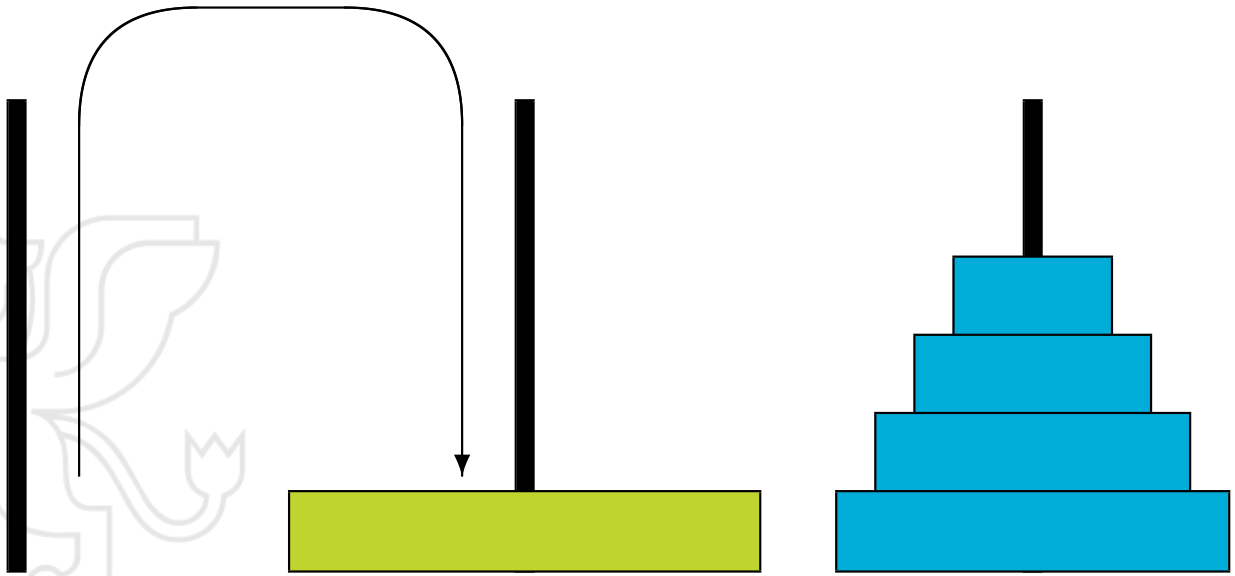
- Az N magasságú torony átpakolását visszavezetjük az N-1 magasságú torony átpakolására.



Hanoi tornyai

Algoritmustervezés 2.

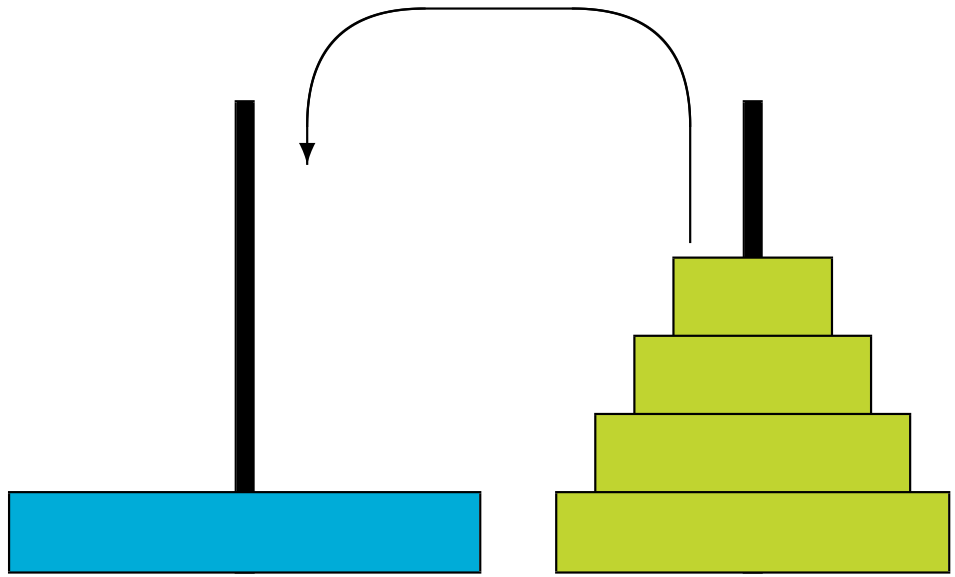
- Az N magasságú torony átpakolását visszavezetjük az N-1 magasságú torony átpakolására.



Hanoi tornyai

Algoritmustervezés 2.

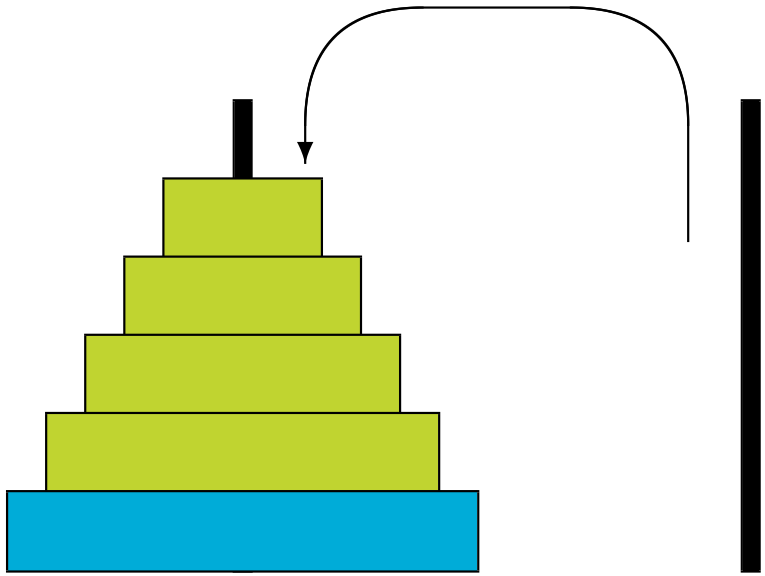
- Az N magasságú torony átpakolását visszavezetjük az N-1 magasságú torony átpakolására.



Hanoi tornyai

Algoritmustervezés 2.

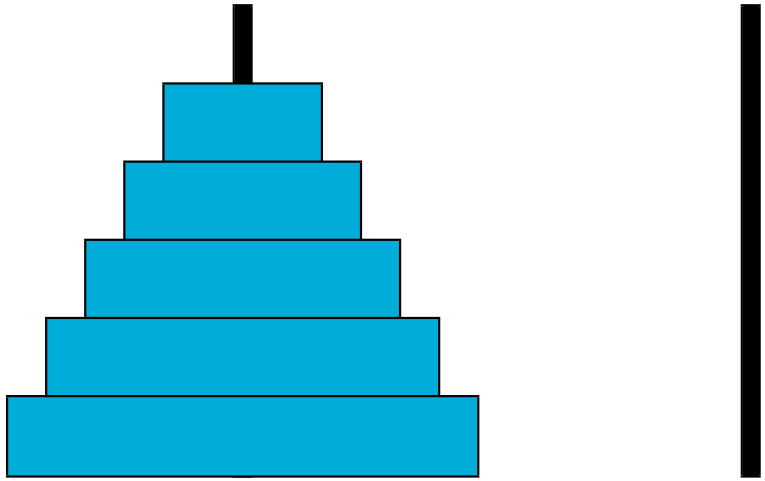
- Az N magasságú torony átpakolását visszavezetjük az N-1 magasságú torony átpakolására.



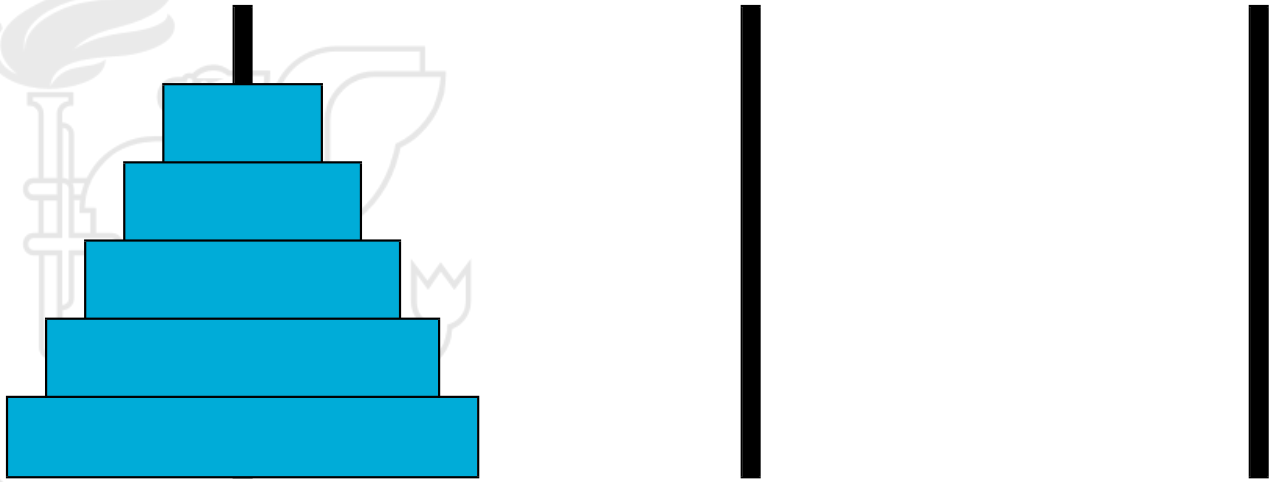
Hanoi tornyai

Algoritmustervezés 2.

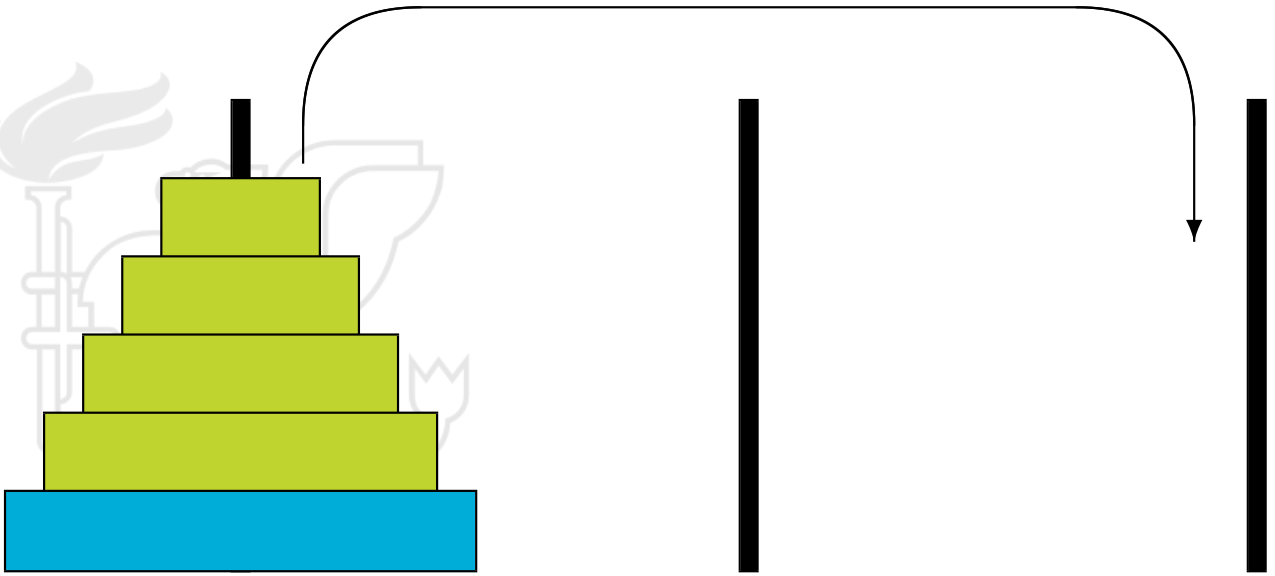
- Az N magasságú torony átpakolását visszavezetjük az N-1 magasságú torony átpakolására.



- Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.

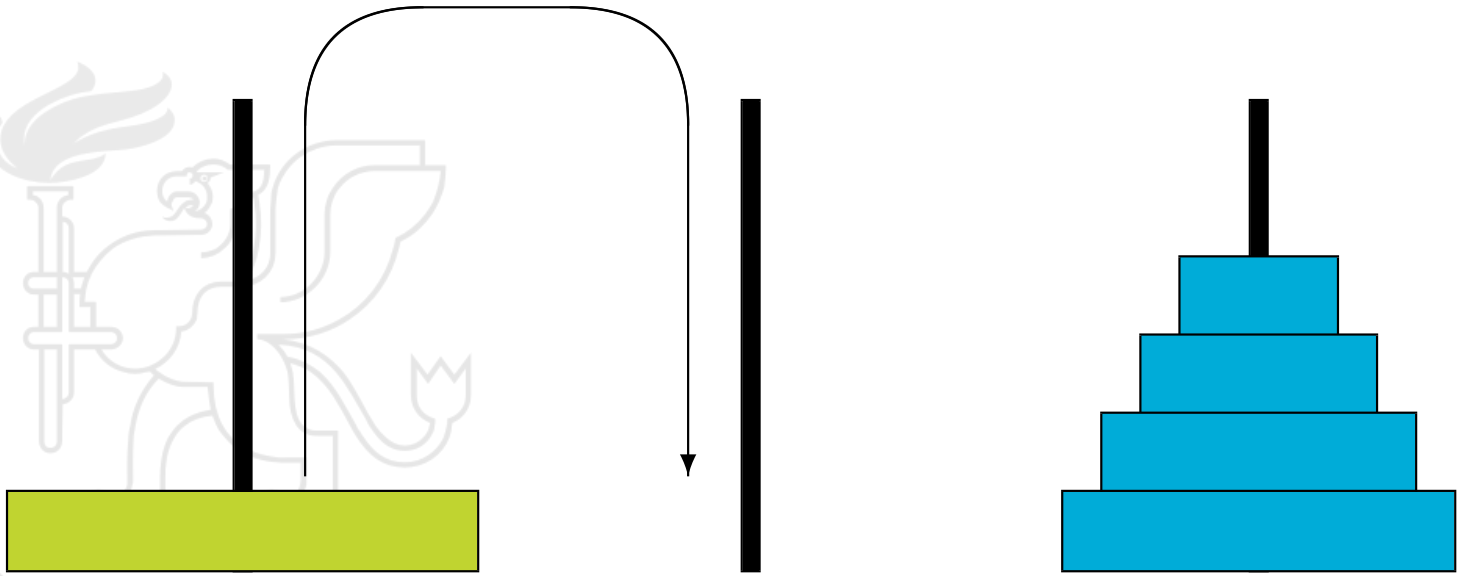


- Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.

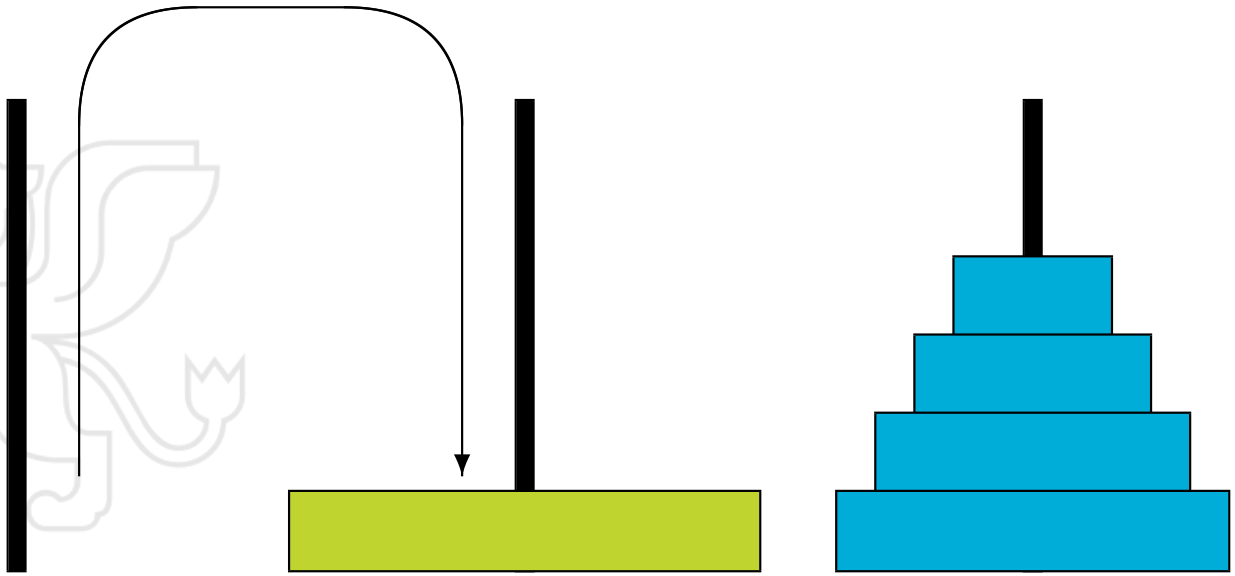


- Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.

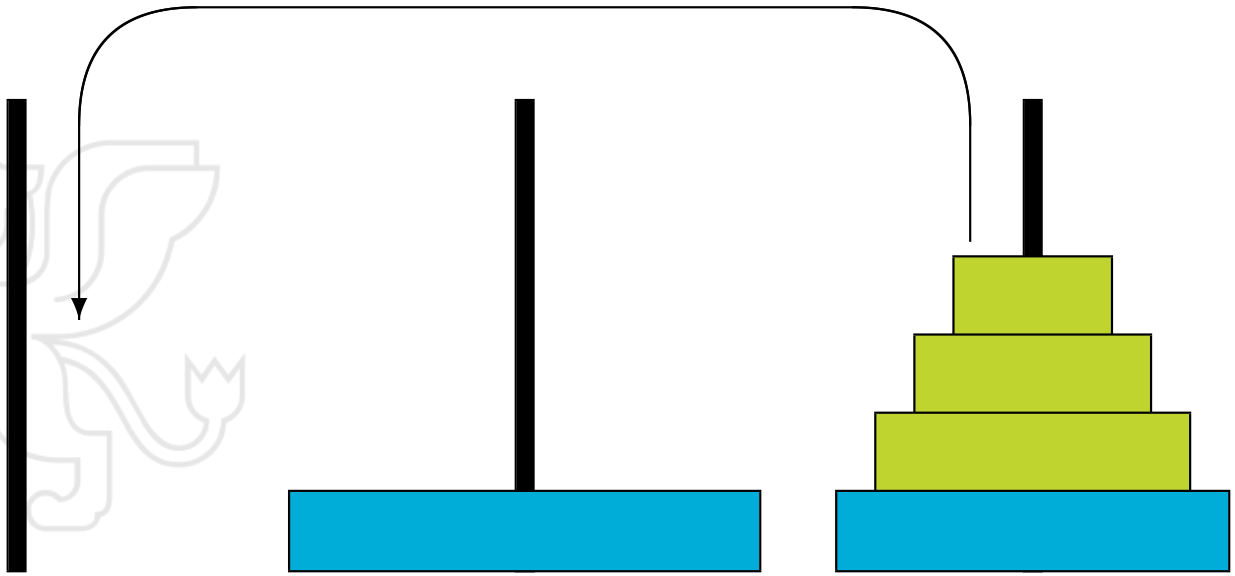
- Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.



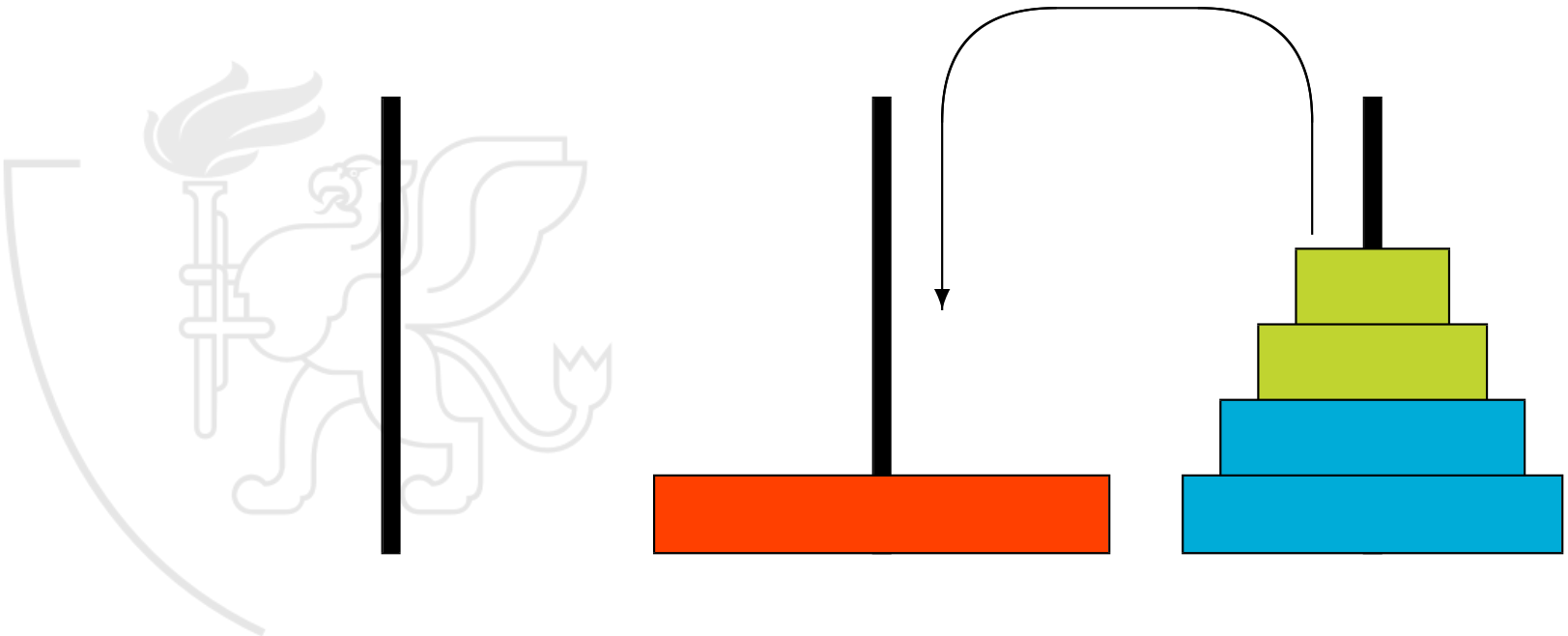
- Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.



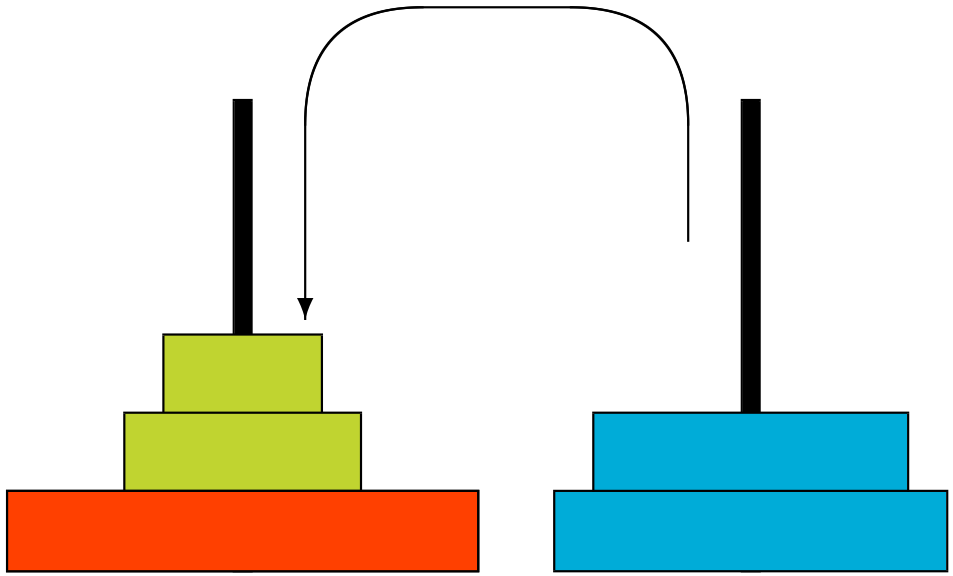
- Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.



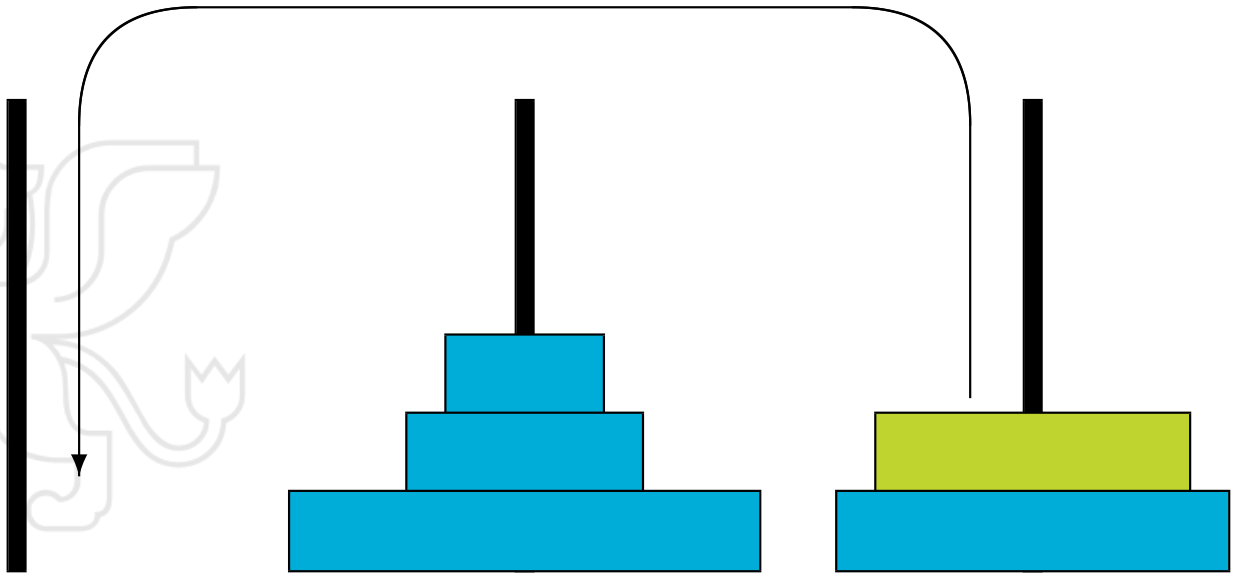
- Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.



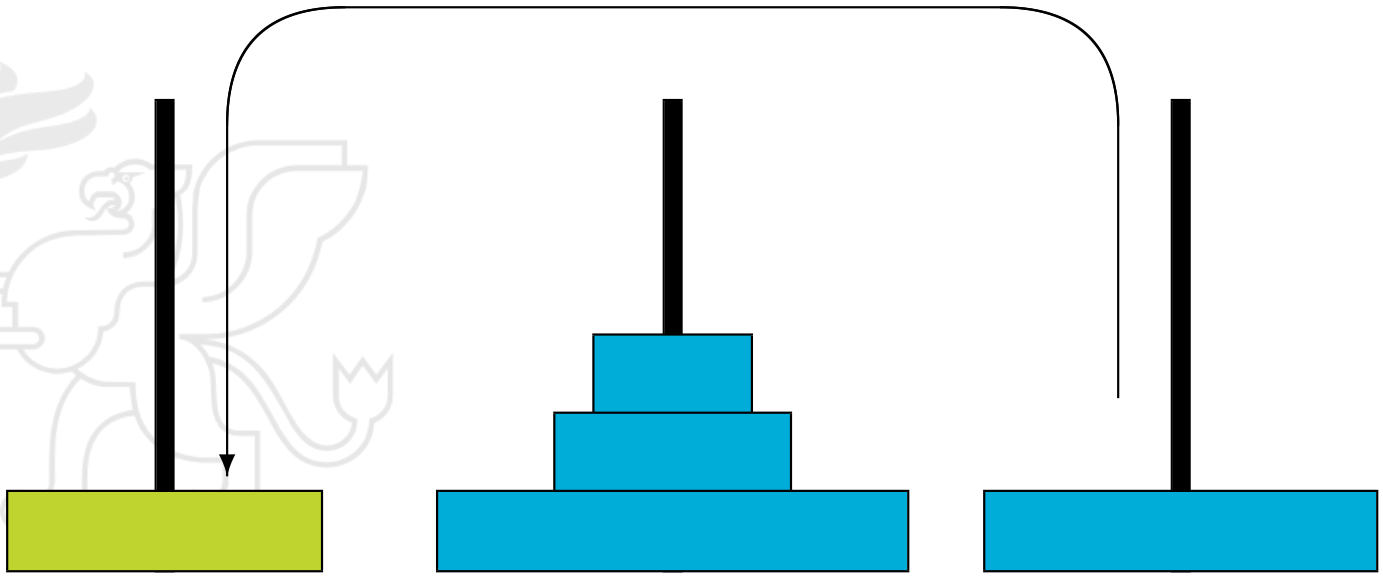
- Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.



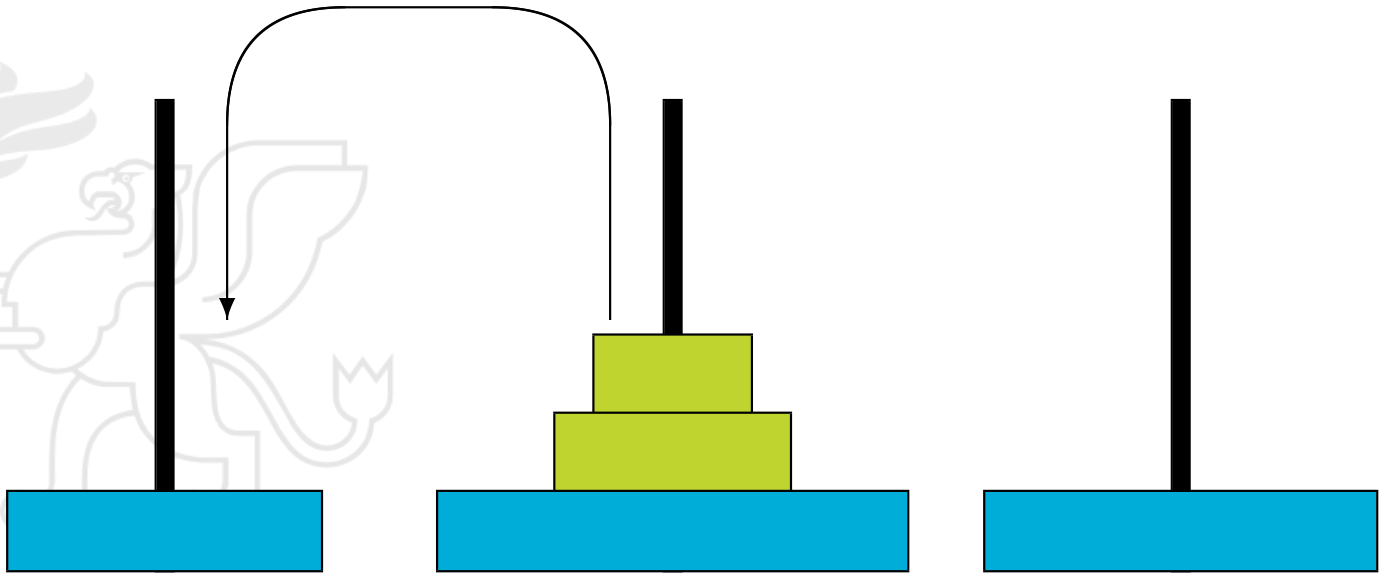
- Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.



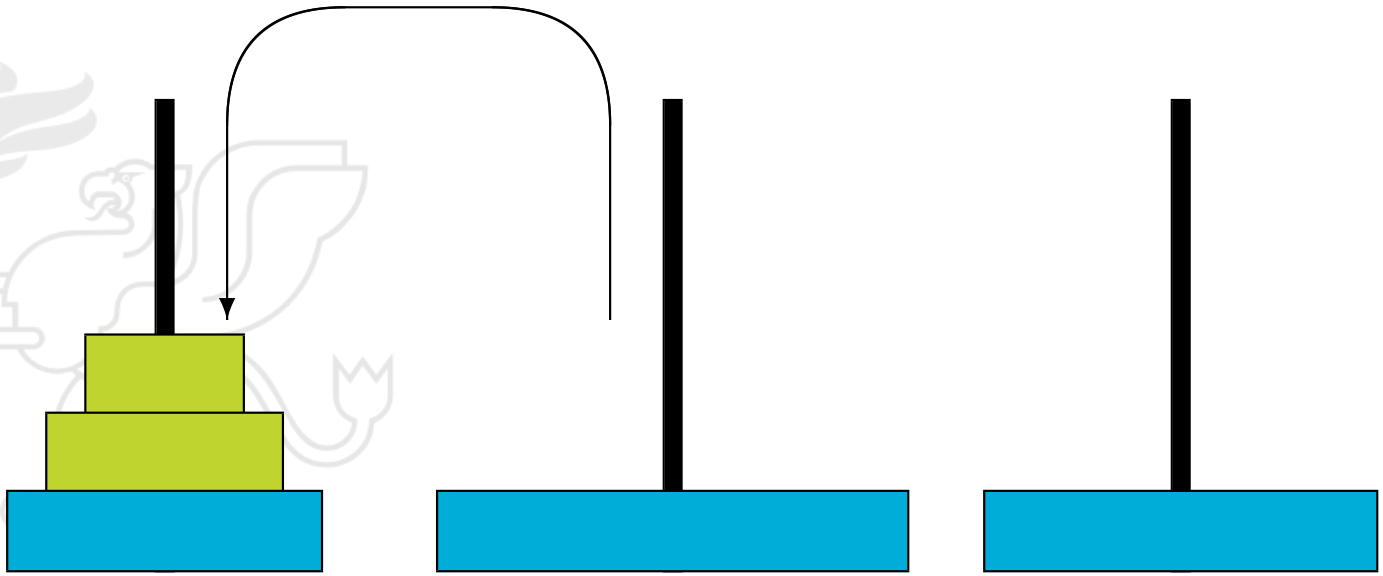
- Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.



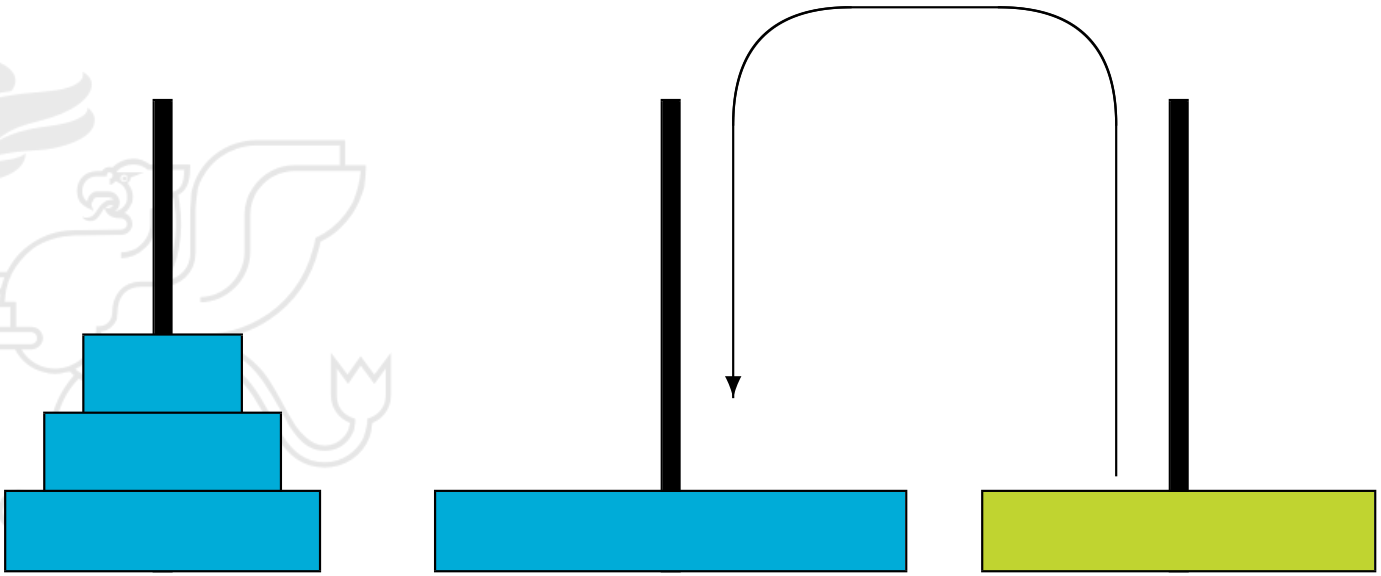
- Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.



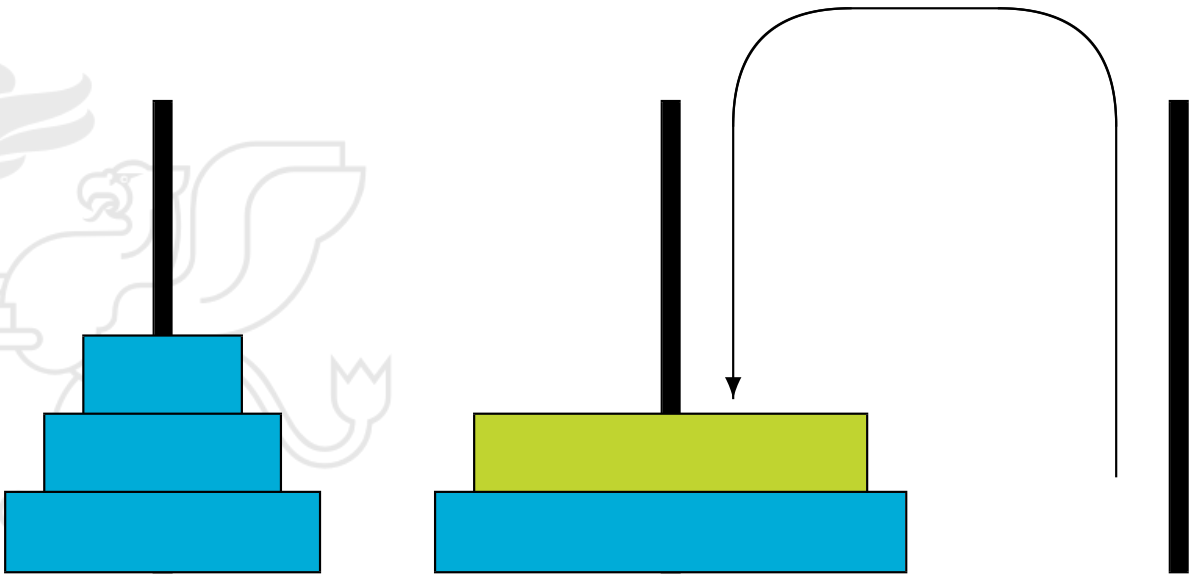
- Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.



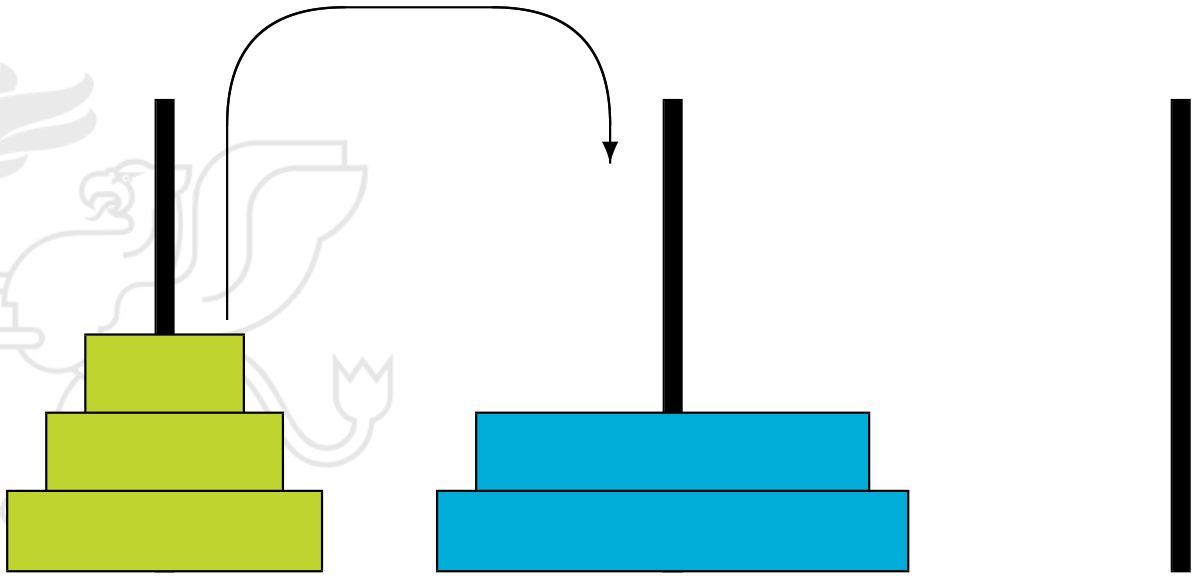
- Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.



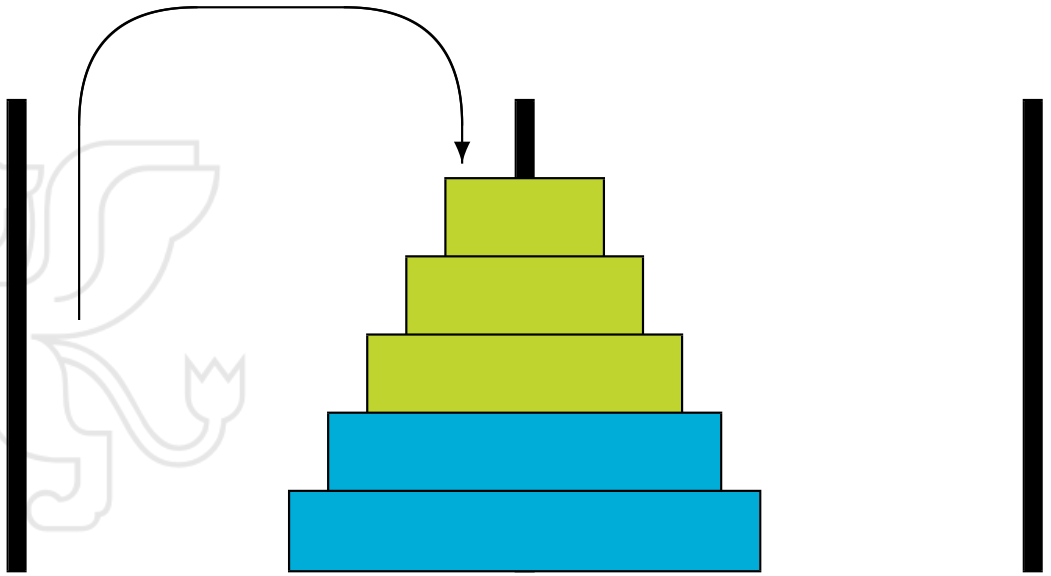
- Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.



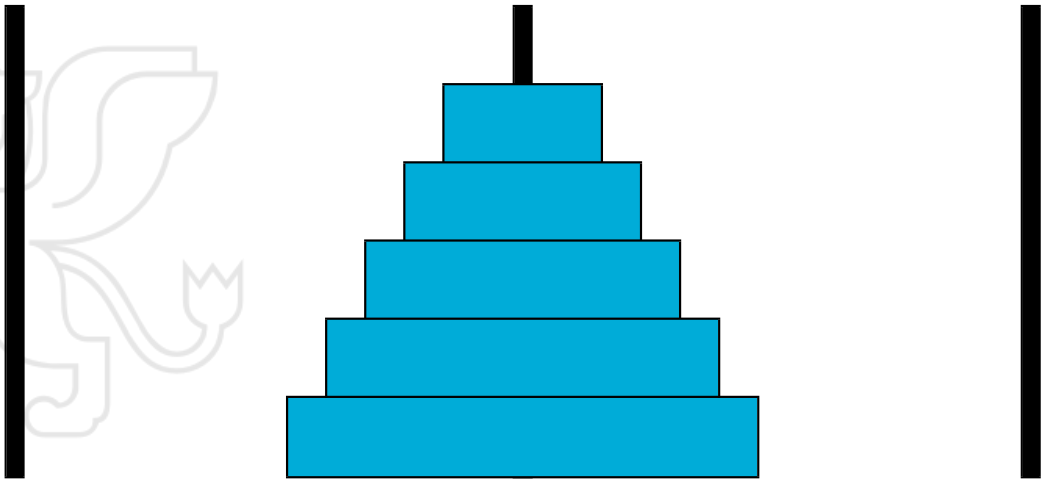
- Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.



- Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.



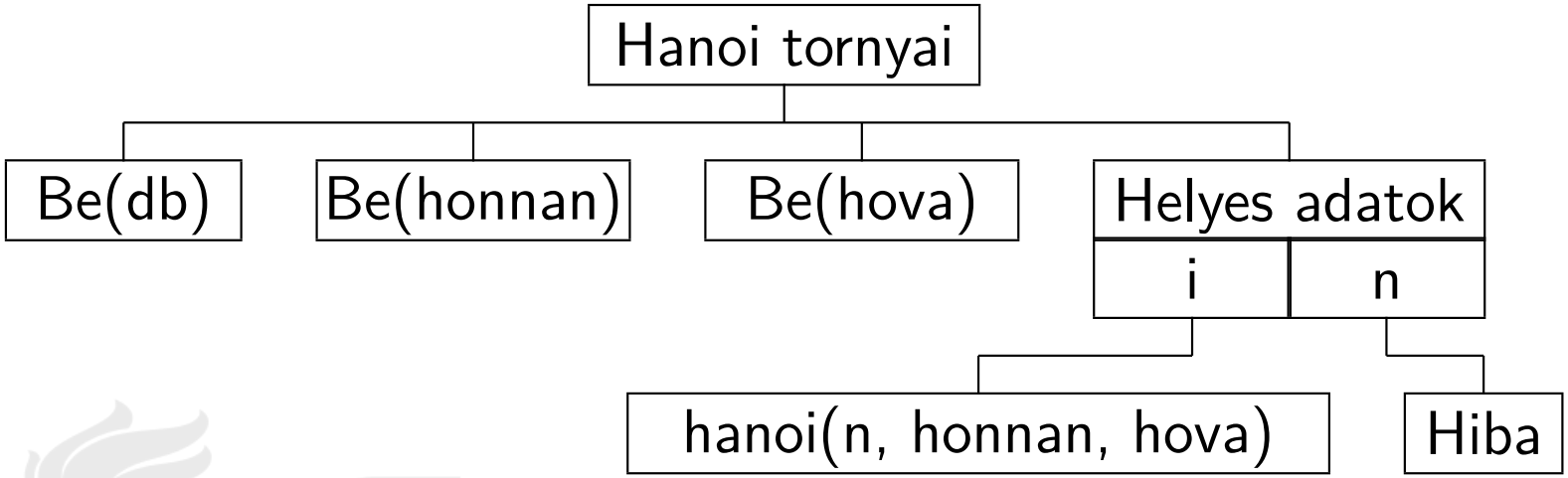
- Mivel az X magasságú torony, amit az egyik rúdról a másikra pakolunk mindig az X legkisebb korongból áll, a harmadik rudat akkor is használhatjuk segédrúdként, ha azon van korong, mivel ez biztosan nagyobb, mint a legnagyobb, amit mi át szeretnénk pakolni.



Hanoi tornyai

Algoritmustervezés – Szerkezeti ábra 1.

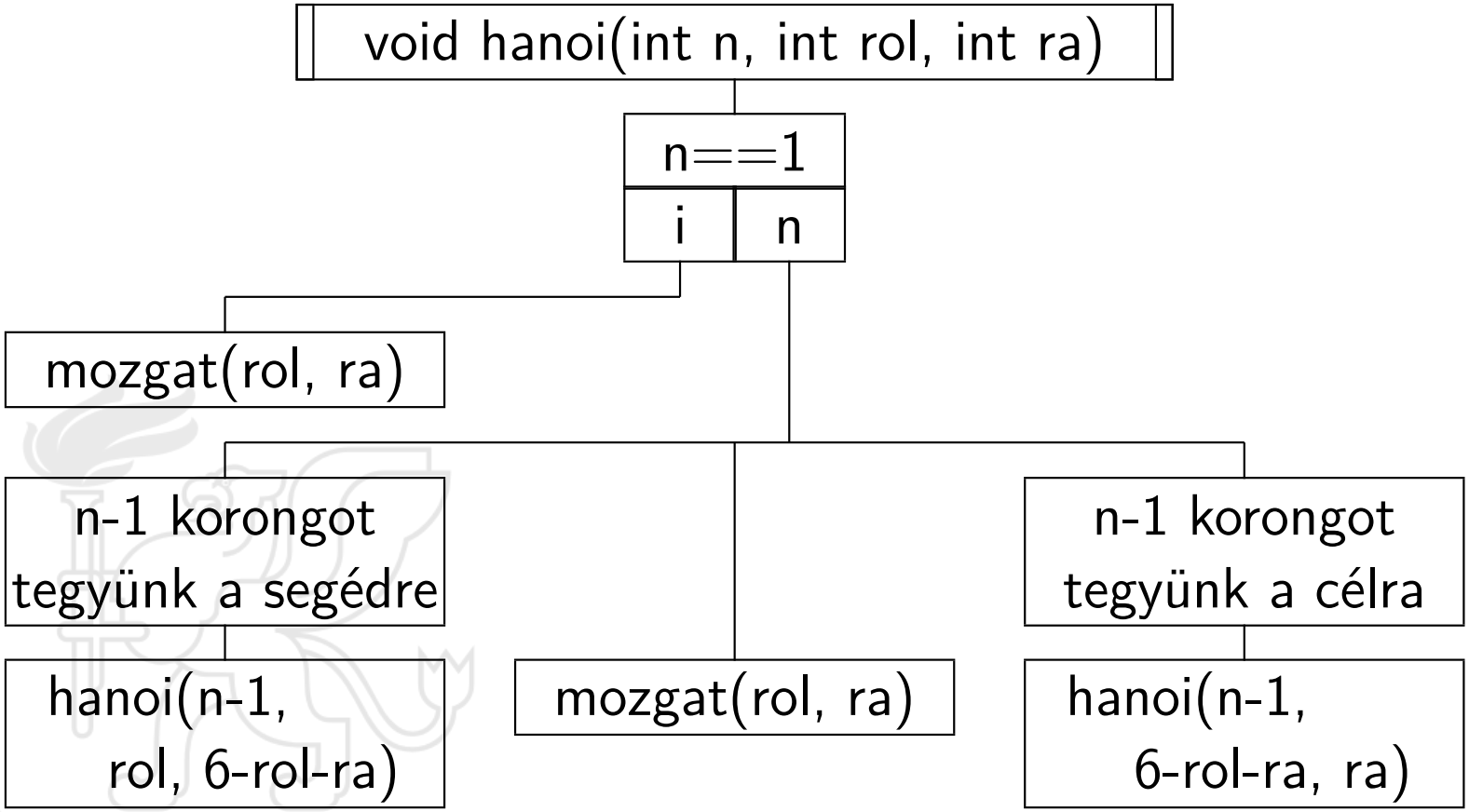
- A főprogram szerkezeti ábrája:



Hanoi tornyai

Algoritmustervezés – Szerkezeti ábra 2.

- A rekurzív függvény szerkezeti ábrája:



Hanoi tornyai [1/2]

hanoi.c [1–21]

```c
/* A Hanoi tornyai játék megvalósítása rekurzív eljárással.
 * 1997. Október 31.  Dévényi Károly, devenyi@inf.u-szeged.hu
 */

#include <stdio.h>

void mozgat(int innen, int ide) {          /* Átrak egy korongot innen ide */
    printf("␣Tegyünk␣át␣egy␣korongot␣a␣%d.␣oszlopról␣a␣%d.␣oszlopra!\n", innen, ide);
}

void hanoi(int n,                                   /* ilyen magas a torony */
           int rol,                                    /* erről a toronyról */
           int ra) {                                     /* erre a toronyra */
    if (n == 1) {
        mozgat(rol, ra);
    } else {
        hanoi(n - 1, rol, 6 - ra - rol);
        mozgat(rol, ra);
        hanoi(n - 1, 6 - ra - rol, ra);
    }
}
```

Hanoi tornyai [2/2]

hanoi.c [23–42]

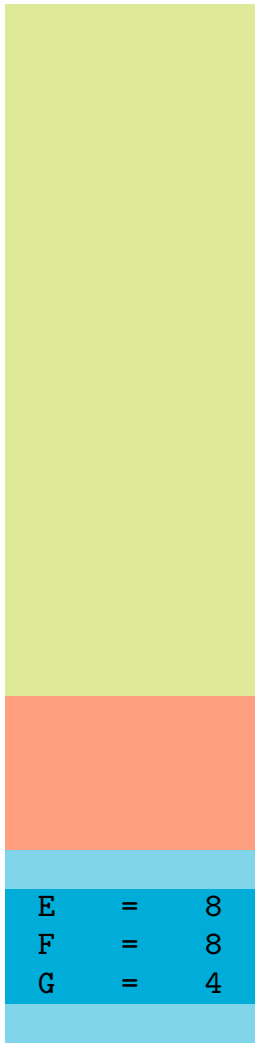
```c
int main() {
    int honnan;                          /* erről a toronyról kell átrakni */
    int hova;                                        /* erre a toronyra */
    int db;                                /* a torony ennyi korongból áll */

    printf("Kérem␣adja␣meg␣a␣torony␣magasságát:␣");
    scanf("%d%*[^\n]", &db); getchar();
    printf("Kérem␣adja␣meg,␣hogy␣a␣torony␣hol␣áll?␣(1,2,3)␣");
    scanf("%d%*[^\n]", &honnan); getchar();
    printf("Kérem␣adja␣meg,␣hogy␣melyik␣oszlopra␣tegyük␣át?␣");
    scanf("%d%*[^\n]", &hova); getchar();
    if (db > 0 &&
        1 <= honnan && honnan <= 3 && 1 <= hova && hova <= 3 &&
        honnan != hova) {
        hanoi(db, honnan, hova);
    } else {
        printf("Hibás␣adat\n");
    }
    return 0;
}
```

Rekurzió

- Az előző példában rekurzív függvénydeklarációt láthattunk, ahol egy függvény saját magát hívta meg.
- A C nyelven bármelyik függvény lehet rekurzív illetve részt vehet rekurzív függvényrendszerben (ahol a függvények nem feltétlenül közvetlenül hívják meg magukat, hanem esetleg más függvényeken keresztül).

Függvényhívás végrehajtása (i386-linux)

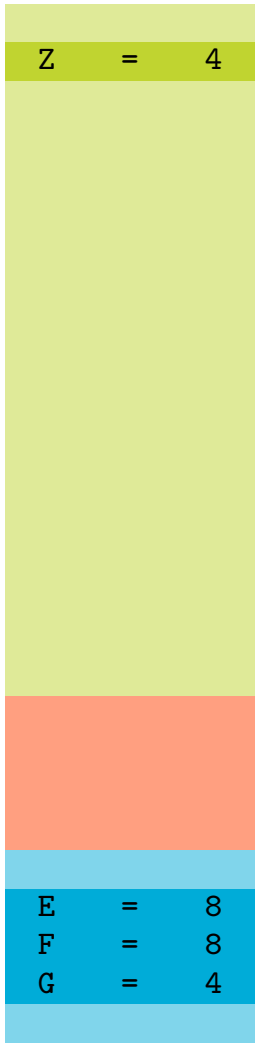
- 0.) A program következő feladata egy függvényhívás kifejezés kiértékelése lesz.




```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
            || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Függvényhívás végrehajtása (i386-linux)

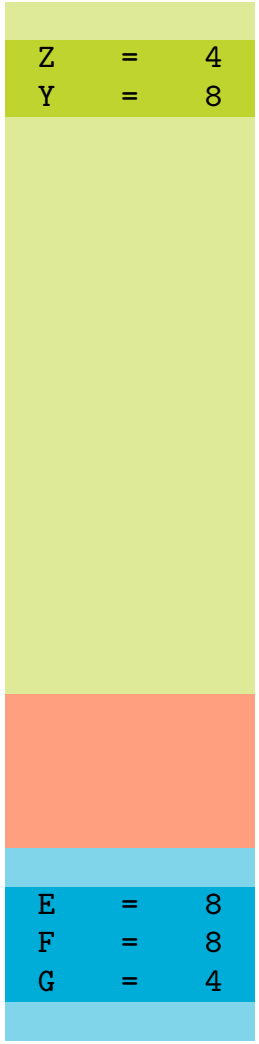
- 1.) Memória helyfoglalás a függvény paraméterei számára és az argumentumok átadása.
  - Az argumentumok elvileg tetszőleges sorrendben kiértékelődhetnek, de a paraméterek jobbról balra sorrendben kerülnek a verembe. Mivel mind értékparaméter, az i-edik argumentum aktuális értéke átadódik az i-edik paraméternek, vagyis az aktuális argumentum értéke bemásolódik a paraméter számára foglalt memóriahelyre.




```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
            || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Függvényhívás végrehajtása (i386-linux)

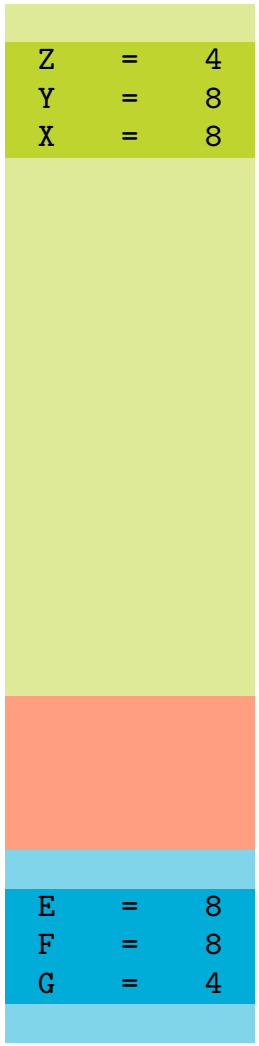
- 1.) Memória helyfoglalás a függvény paraméterei számára és az argumentumok átadása.
  - Az argumentumok elvileg tetszőleges sorrendben kiértékelődhetnek, de a paraméterek jobbról balra sorrendben kerülnek a verembe. Mivel mind értékparaméter, az i-edik argumentum aktuális értéke átadódik az i-edik paraméternek, vagyis az aktuális argumentum értéke bemásolódik a paraméter számára foglalt memóriahelyre.




```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
            || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Függvényhívás végrehajtása (i386-linux)

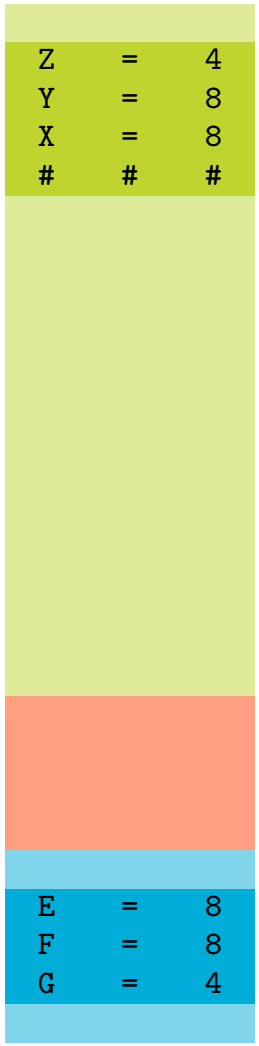
- 1.) Memória helyfoglalás a függvény paraméterei számára és az argumentumok átadása.
  - Az argumentumok elvileg tetszőleges sorrendben kiértékelődhetnek, de a paraméterek jobbról balra sorrendben kerülnek a verembe. Mivel mind értékparaméter, az i-edik argumentum aktuális értéke átadódik az i-edik paraméternek, vagyis az aktuális argumentum értéke bemásolódik a paraméter számára foglalt memóriahelyre.




```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
            || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Függvényhívás végrehajtása (i386-linux)

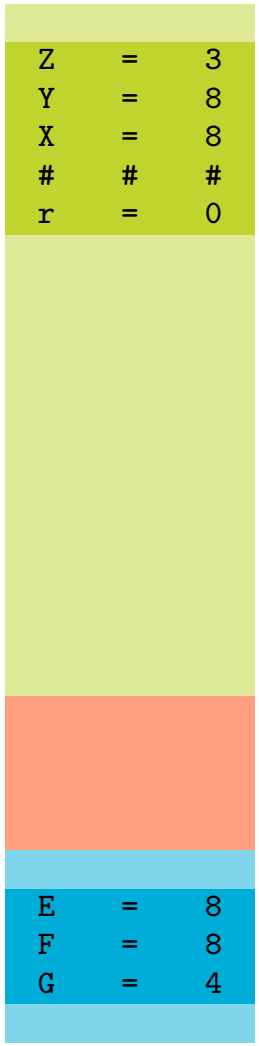
- 2.) Függvényhívás
  - A verembe bekerülnek a technikai információk (pl. visszatérési cím), és a vezérlés átadódik a függvénynek.




```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
            || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Függvényhívás végrehajtása (i386-linux)

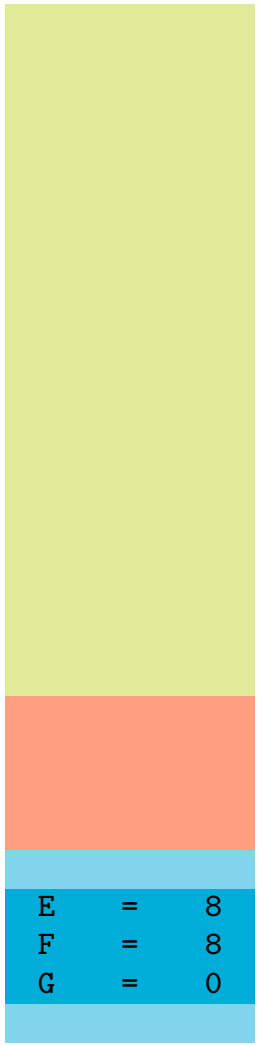
- 3.) A függvényblokk utasításrészének végrehajtása.
  - Elkezdjük végrehajtani a függvényblokkot, egészen a `return` utasításig.




```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
            || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Függvényhívás végrehajtása (i386-linux)

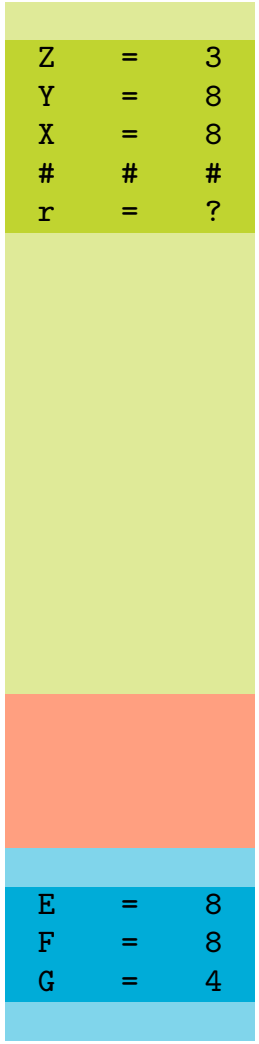
- 4.) Visszatérés.
  - A függvényblokk formális paraméterei és lokális változói számára foglalt memória felszabadítása.
  - A függvényhívás kifejezése felveszi a függvényben kiszámolt értéket.




```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
            || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Függvényhívás végrehajtása rekurzióval (i386-linux)

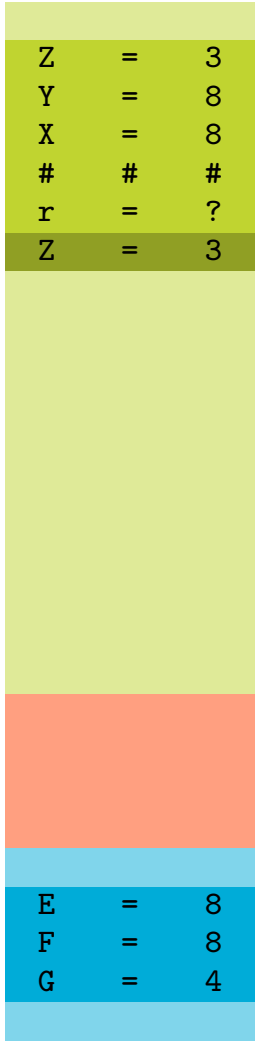
- 0.) A program következő feladata egy függvényhívás kifejezés kiértékelése lesz.
  - A rekurzió is „csak egy” függvényhívás, tehát pontosan ugyanazokat a lépéseket kell végrehajtani. Annak, hogy egy már végrehjtás alatt lévő függvényt (rekurzió) vagy másik függvényt hívunk, a függvényhíváskor végrehajtott műveletek tekintetében semmi jelentőssége nincs.




```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
          || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Függvényhívás végrehajtása rekurzióval (i386-linux)

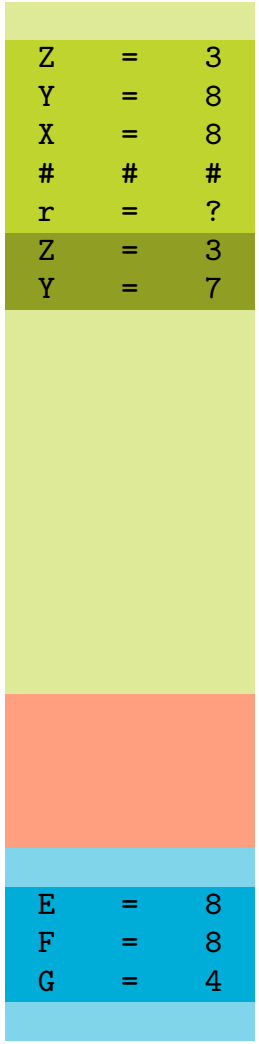
- 1.) Memória helyfoglalás a függvény paraméterei számára és az argumentumok átadása.
  - Az argumentumok elvileg tetszőleges sorrendben kiértékelődhetnek, de a paraméterek jobbról balra sorrendben kerülnek a verembe. Mivel mind értékparaméter, az i-edik argumentum aktuális értéke átadódik az i-edik paraméternek, vagyis az aktuális argumentum értéke bemásolódik a paraméter számára foglalt memóriahelyre.




```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
            || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Függvényhívás végrehajtása rekurzióval (i386-linux)

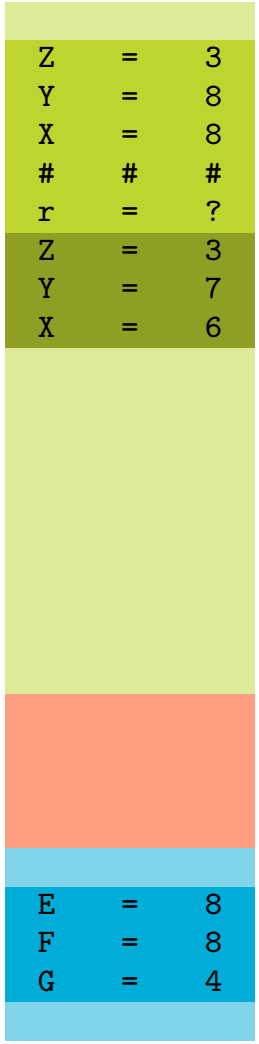
- 1.) Memória helyfoglalás a függvény paraméterei számára és az argumentumok átadása.
  - Az argumentumok elvileg tetszőleges sorrendben kiértékelődhetnek, de a paraméterek jobbról balra sorrendben kerülnek a verembe. Mivel mind értékparaméter, az i-edik argumentum aktuális értéke átadódik az i-edik paraméternek, vagyis az aktuális argumentum értéke bemásolódik a paraméter számára foglalt memóriahelyre.




```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
          || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Függvényhívás végrehajtása rekurzióval (i386-linux)

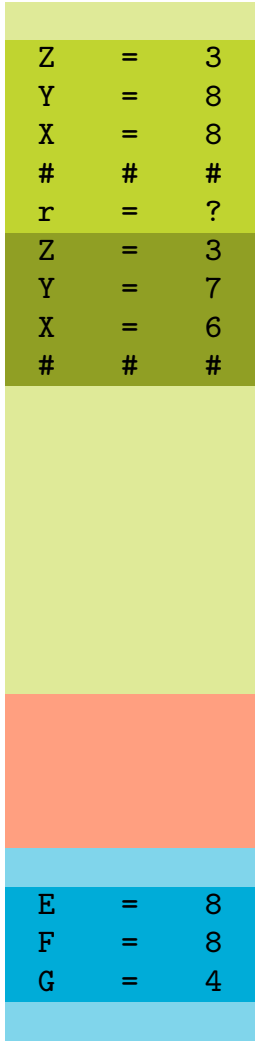
- 1.) Memória helyfoglalás a függvény paraméterei számára és az argumentumok átadása.
  - Az argumentumok elvileg tetszőleges sorrendben kiértékelődhetnek, de a paraméterek jobbról balra sorrendben kerülnek a verembe. Mivel mind értékparaméter, az i-edik argumentum aktuális értéke átadódik az i-edik paraméternek, vagyis az aktuális argumentum értéke bemásolódik a paraméter számára foglalt memóriahelyre.




```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
          || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Függvényhívás végrehajtása rekurzióval (i386-linux)

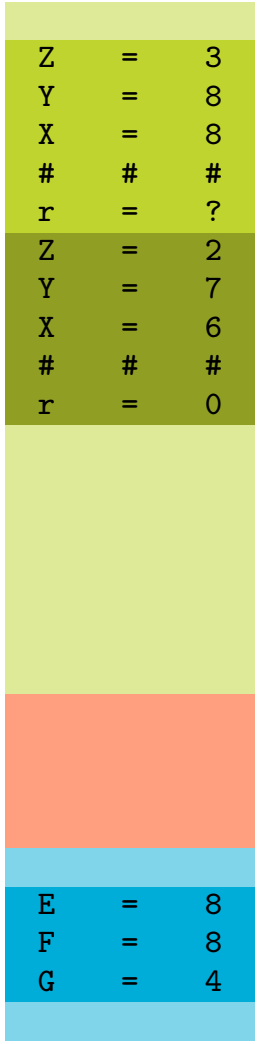
- 2.) Függvényhívás
  - A verembe bekerülnek a technikai információk (pl. visszatérési cím), és a vezérlés átadódik a függvénynek.




```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
          || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Függvényhívás végrehajtása rekurzióval (i386-linux)

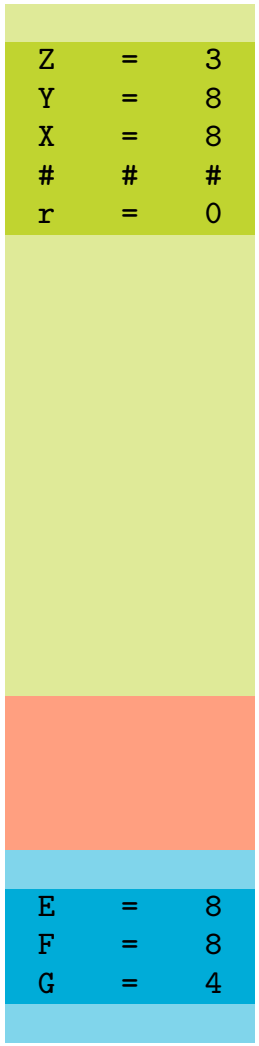
- 3.) A függvényblokk utasításrészének végrehajtása.
  - Elkezdjük végrehajtani a függvényblokkot, egészen a `return` utasításig.




```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
            || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Függvényhívás végrehajtása rekurzióval (i386-linux)

- 4.) Visszatérés.
  - A függvényblokk formális paraméterei és lokális változói számára foglalt memória felszabadítása.
  - A függvényhívás kifejezése felveszi a függvényben kiszámolt értéket.



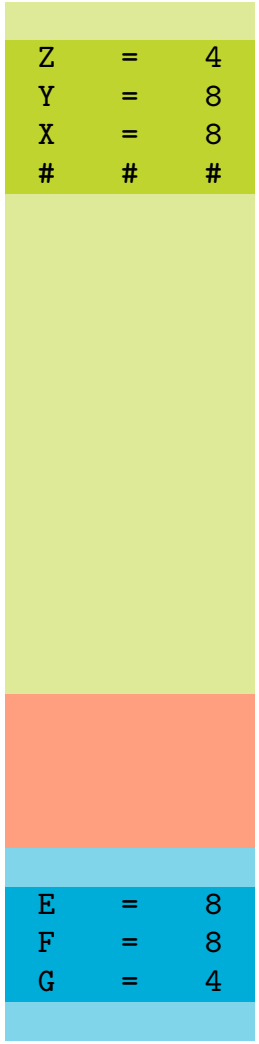

```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
          || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Blokkok végrehajtása

- A C nyelvben nem csak függvény szinten, hanem blokk szinten lehet változókat deklarálni.
- Ezek tárolását a C szintén a veremben végzi. Emiatt egy blokk végrehajtásának lépései nagyon hasonlítanak a függvények végrehajtásához.

Blokkok végrehajtása

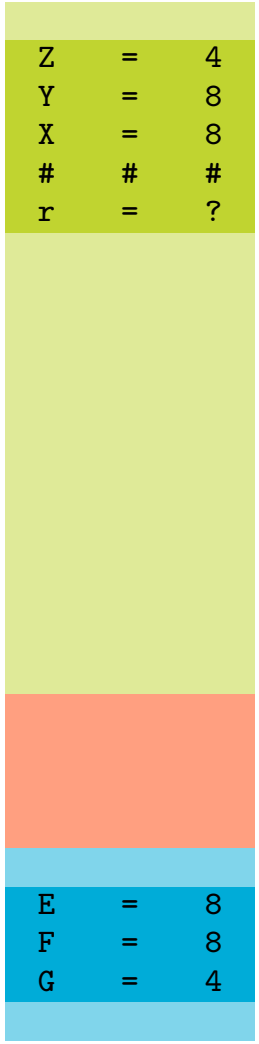
- 0.) A program következő feladata egy blokk végrehajtása lesz.
  - A C nyelvben nem csak függvény szinten, hanem blokk szinten lehet változókat deklarálni.
  - Ezek tárolását a C szintén a veremben végzi. Emiatt egy blokk végrehajtásának lépései nagyon hasonlítanak a függvények végrehajtásához.




```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
            || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Blokkok végrehajtása

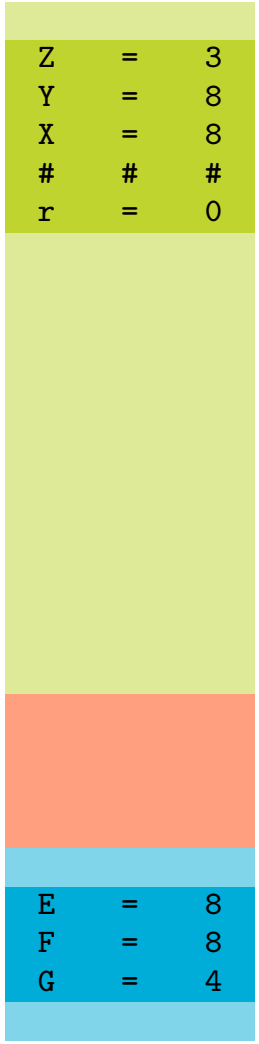
- 1.) Memória helyfoglalás a blokk változói számára.




```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
            || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Blokkok végrehajtása

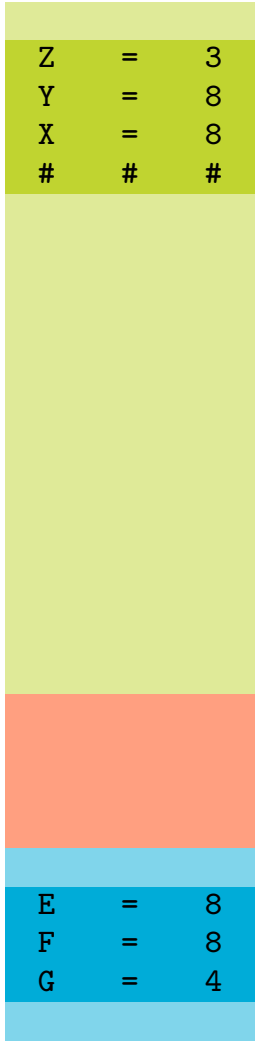
- 2.) A blokk végrehajtása.




```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
            || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Blokkok végrehajtása

- 3.) Memória felszabadítása.




```c
#include <stdio.h>
int E, F, G;
int A(int X, int Y, int Z) {
  int r;
  if (Z < 0 || X < 1 || Y < 1
            || 8 < X || 8 < Y)
    return 0;
  if (X == 1 && Y == 1)
    return 1;
  --Z;
  r = A(X - 2, Y - 1, Z);
  r = r || A(X - 2, Y + 1, Z);
  r = r || A(X - 1, Y - 2, Z);
  r = r || A(X - 1, Y + 2, Z);
  r = r || A(X + 1, Y - 2, Z);
  r = r || A(X + 1, Y + 2, Z);
  r = r || A(X + 2, Y - 1, Z);
  r = r || A(X + 2, Y + 1, Z);
  return r;
}

int main() {
  scanf("%d%d%d", &E, &F, &G);
  G = A(E, F, G);
  printf(G ? "+\n" : "-\n");
  return 0;
}
```

Függvények mellékhatása

- Függvény mellékhatásán azt értjük, hogy a függvényhívás hatására nem csak a függvényérték számítódik ki (és a paraméterek változhatnak meg), hanem megváltozhat egy globális változó értéke is (vagy egyéb műveletek is végrehajtódnak, pl. kiíratás).
- Mellékhatás következménye, hogy az összeadás kommutativitása nem feltétlenül teljesül, ha a tagok függvényhívások.
- C-ben ugyanis nincs meghatározva, hogy két részkifejezés közül melyiket kell előbb kiértékelni, tehát az sem világos, hogy ha mindkettőben van függvényhívás, melyik hajtódik végre előbb.

```c
int A, B, Z;
int f(int x) {
    int r;
    r = x + A;
    A = x + 1;
    return r;
}
int main () {
    A = 1; B = 2;
    Z = f(A) + f(B);  /* Z==6?  Z==9? */
    return 0;
}
```

Függvények előnyei

- Többszörös felhasználás: Hasonló részproblémák megoldására elég egy függvényt készíteni és a különböző adatokra végrehajtatni a részalgoritmust. Így a program méretét csökkenteni lehet.
- Memória igény csökkentése: A függvények lokális változói számára csak a függvény végrehajtása idejére foglalódik memória.
- Karbantarthatóság: Függvények használatával a program áttekinthetőbb lesz, ami jelentősen megkönyíti a későbbi módosítását.
- Modularitás: A tervezés során a részproblémák függvénnyel történő megoldása lehetővé teszi a figyelem lokalizálását.
- Programhelyesség: Függvények alkalmazása megkönnyíti a bizonyítást, a program tesztelését, a hibakeresést és javítást.

Tartalom

$\binom{n}{k}$ kiszámítása

Problémafelvetés és specifikáció

- Problémafelvetés
  - Számítsuk ki $\binom{n}{k}$ értékét
- Specifikáció
  - A probléma bemenete két egész szám, n és k.
  - A probléma kimenete $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ ha értelmezhető n-re és k-ra, különben 0.

- Készítsünk az $\binom{n}{k}$ kiszámolására egy függvényt.
- Ismeretes, hogy $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, de az is nyilvánvaló, hogy nem célszerű a számolást a faktoriálisok számításával kezdeni majd az osztás és szorzás műveletet ezekre elvégezni. Így ugyanis nagyon hamar kifutnánk a C egész típusok értékkészletéből ($21! > 2^{63}$).
- A fenti képletet egyszerűsítve és átalakítva kapjuk, hogy

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n \cdot (n-1) \cdot \dots \cdot (n-k+1) \cdot (n-k) \cdot \dots \cdot 1}{k \cdot (k-1) \cdot \dots \cdot 1 \cdot (n-k) \cdot \dots \cdot 1}$$

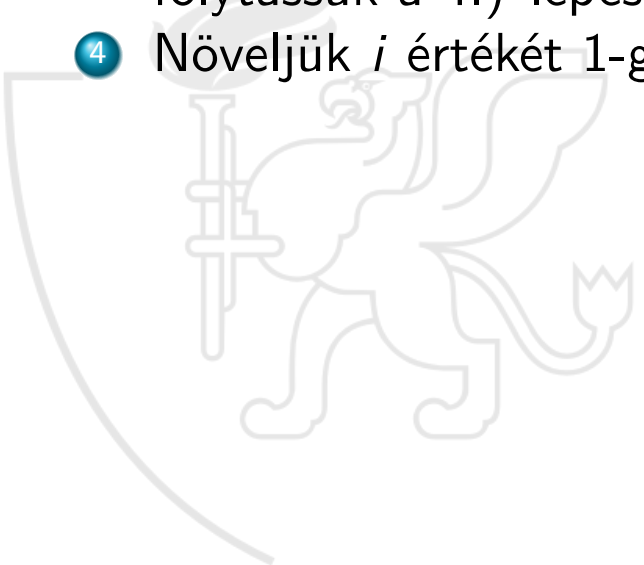
$$= \frac{n \cdot (n-1) \cdot \dots \cdot (n-k+1)}{1 \cdot 2 \cdot \dots \cdot k} = \prod_{i=1}^{k} \frac{n-i+1}{i}$$

Számlálásos ismétléses vezérlések

- Számlálásos ismétléses vezérlésről akkor beszélünk, ha a ciklusmagot végre kell hajtani egy változó sorban minden olyan értékére (növekvő vagy csökkenő sorrendben), amely egy adott intervallumba esik.
- Legyen a és b egész érték, i egész típusú változó, M pedig tetszőleges művelet, amelynek nincs hatása a, b és i értékére.

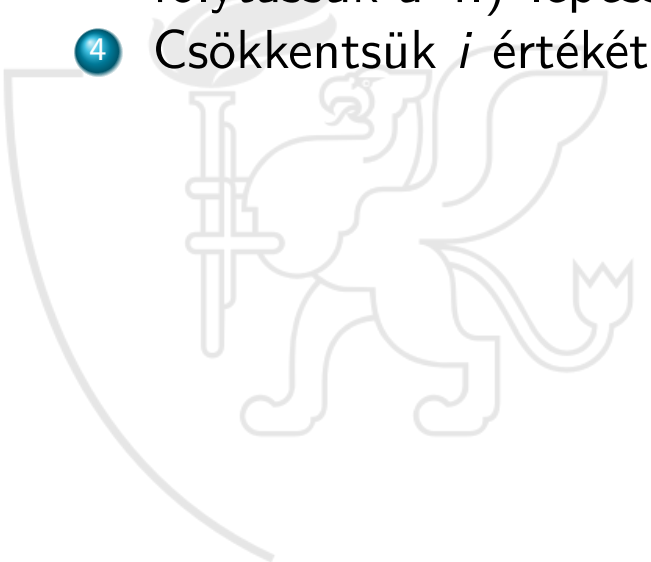
Növekvő számlálásos ismétléses vezérlés

- Az a és b határértékekből, i ciklusváltozóból és M műveletből (ciklusmagból) képzett növekvő számlálásos ismétléses vezérlés az alábbi vezérlési előírást jelenti:
  1. Legyen $i = a$ és folytassuk a 2.) lépéssel.
  2. Ha $b < i$, akkor az ismétlés és ezzel együtt az összetett művelet végrehajtása befejeződött.
  3. Egyébként, vagyis ha $i \leq b$, akkor hajtsuk végre az M műveletet, majd folytassuk a 4.) lépéssel.
  4. Növeljük i értékét 1-gyel, és folytassuk a 2.) lépéssel.



Csökkenő számlálásos ismétléses vezérlés

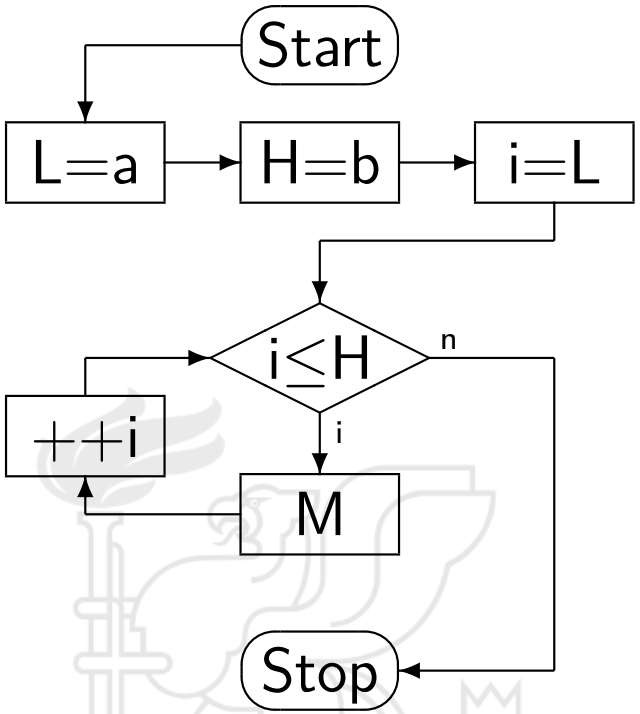
- Az a és b határértékekből, i ciklusváltozóból és M műveletből (ciklusmagból) képzett csökkenő számlálásos ismétléses vezérlés az alábbi vezérlési előírást jelenti:
  1. Legyen $i = b$ és folytassuk a 2.) lépéssel.
  2. Ha $i < a$, akkor az ismétlés és ezzel együtt az összetett művelet végrehajtása befejeződött.
  3. Egyébként, vagyis ha $a \leq i$, akkor hajtsuk végre az M műveletet, majd folytassuk a 4.) lépéssel.
  4. Csökkentsük i értékét 1-gyel, és folytassuk a 2.) lépéssel.



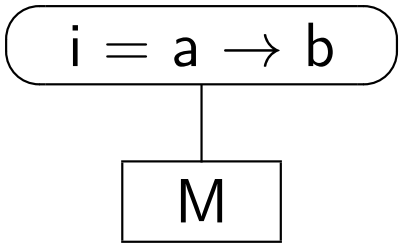
Növekvő számlálásos ismétléses vezérlés

Folyamatábra és szerkezeti ábra

- Folyamatábrája:



- Szerkezeti ábrája:

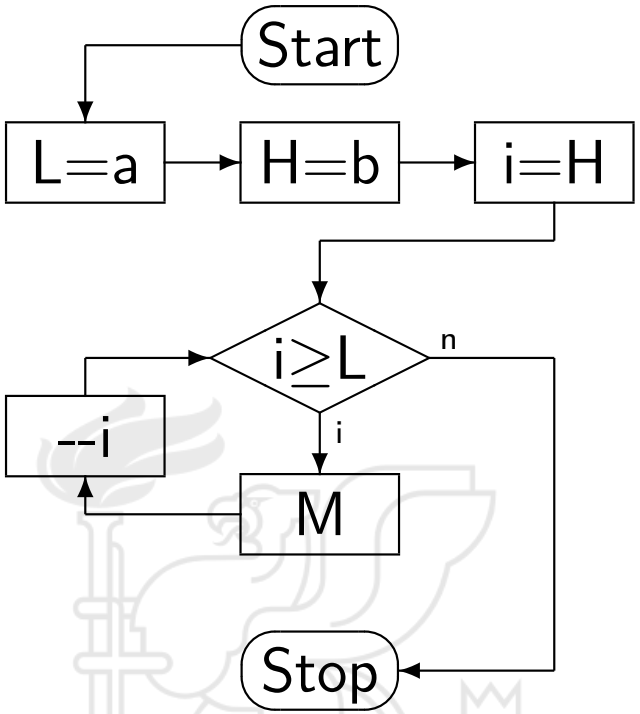


- A határértékek alapján előfordulhat, hogy az *M* műveletet egyszer sem hajtjuk végre.

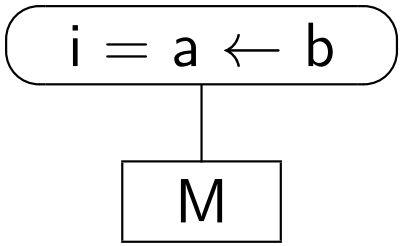
Csökkenő számlálásos ismétléses vezérlés

Folyamatábra és szerkezeti ábra

- Folyamatábrája:



- Szerkezeti ábrája:

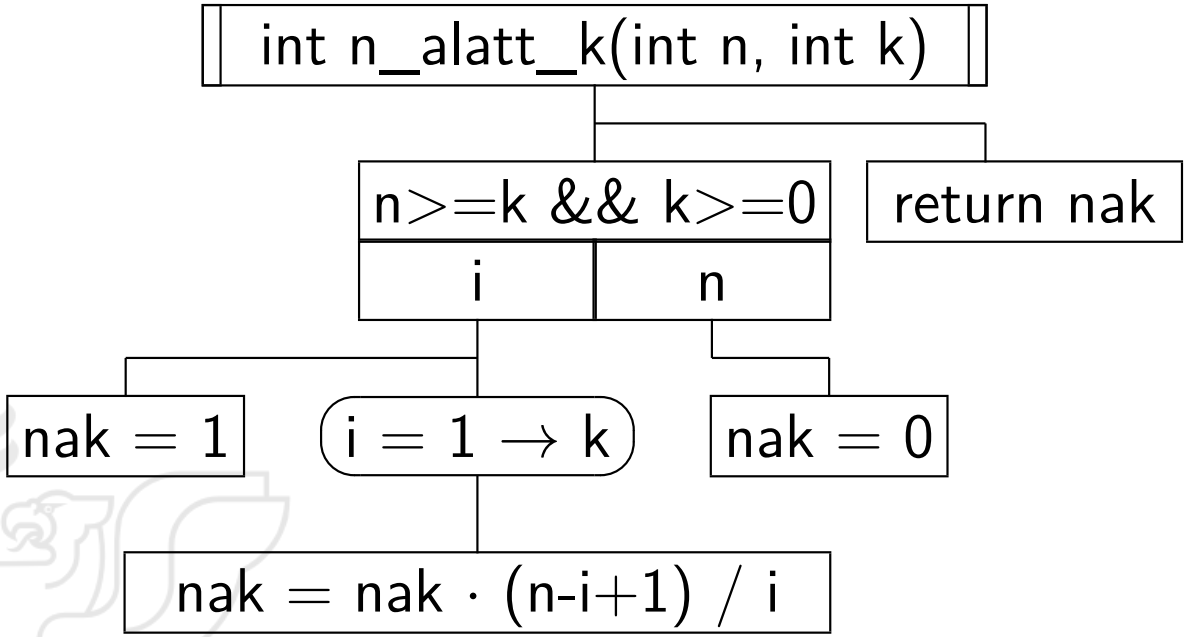


- A határértékek alapján előfordulhat, hogy az *M* műveletet egyszer sem hajtjuk végre.

$\binom{n}{k}$ kiszámítása

Algoritmustervezés – Szerkezeti ábra

- Az $\binom{n}{k}$ értékét számító függvény szerkezeti ábrája:



A `for` utasítás

- Ha valamilyen műveletet sorban több értékére is végre kell hajtani, akkor a `for` utasítás (is) használható.
- A `for` utasítás szintaxisa C-ben

A `for` utasítás

- C-ben a `for` utasítás általános alakja így néz ki:

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

ami egyenértékű ezzel:

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

- A $\texttt{kif}_1$ és $\texttt{kif}_3$ többnyire értékadás vagy függvényhívás, $\texttt{kif}_2$ pedig relációs kifejezés.
- Bármelyik kifejezés elhagyható, de a pontosvesszőknek meg kell maradniuk. $\texttt{kif}_2$ elhagyása esetén a feltételt konstans igaznak tekintjük, ekkor a `break` vagy `return` segítségével lehet kiugrani a ciklusból.

A , művelet

- A , egy különleges művelet C-ben, amely balról jobbra kiértékeli a két részkifejezését, majd a művelet eredménye a jobboldali részkifejezés értéke lesz.
- A , operátor sorozatos alkalmazása így tulajdonképpen szekvenciális vezérlést ír elő egyetlen kifejezésen belül.
- A művelet leggyakoribb használata a `for` ciklushoz kapcsolódik:
  - Előfordul, hogy az inicializáló és az inkrementáló rész is több adminisztratív lépést tartalmaz.
  - A , művelet segítségével ezeket egy-egy kifejezéssé tudjuk összeolvasztani. Ilyen módon az alábbi két kódrészlet ekvivalens:

```
for (kif1,1, kif1,2, kif1,3;
     kif2;
     kif3,1, kif3,2, kif3,3) {
    utasítás;
}
```

```
kif1,1; kif1,2; kif1,3;
while (kif2) {
    utasítás;
    kif3,1; kif3,2; kif3,3;
}
```

C műveletek prioritása

műveletek	asszoc.	C operátorok
egyoperandusú postfix, elemkiválasztás, mezőkiválasztás, függvényhívás	→	`x++`, `x--`, `[]`, `.`, `->`, `f()`
egyoperandusú prefix, méret, típuskényszerítés	←	`+`, `-`, `++x`, `--x`, `!`, `~`, `&`, `*`, `sizeof`, `(type)`
multiplikatív	→	`*`, `/`, `%`
additív	→	`+`, `-`
bitléptetés	→	`<<`, `>>`
kisebb-nagyobb relációs	→	`<=`, `>=`, `<`, `>`
egyenlő-nem egyenlő relációs	→	`==`, `!=`
bitenkénti 'és'	→	`&`
bitenkénti 'kizáró vagy'	→	`^`
bitenkénti 'vagy'	→	`|`
logikai 'és'	→	`&&`
logikai 'vagy'	→	`||`
feltételes	←	`?:`
értékadó	←	`=`, `+=`, `-=`, `*=`, `/=`, `%=` `>>=`, `<<=`, `&=`, `^=`, `|=`
szekvencia	→	`,`

Ábécé

Problémafelvetés, specifikáció, algoritmustervezés

- Problémafelvetés
  - Írassuk ki felváltva az angol ábécé kis és nagybetűit.
- Specifikáció
  - A problémának nincs bemenete, a kimenete pedig az angol ábécé 26 kis- és 26 nagybetűje felváltva kiíratva.
- Algoritmustervezés
  - Egy számlálásos vezérlés kell az angol ábécé 26 betűjére, amely egy ciklusléptésben kiírja ugyanazon betű kis és nagy változatát is.

ABC [1/1]

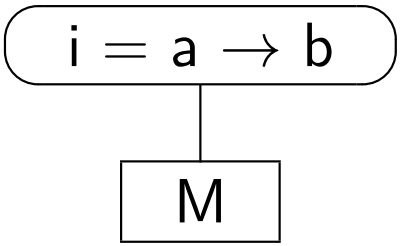
abc.c [1–17]

```c
/* Az ábécé kis- és nagybetűinek kiíratása összefésülve.
 * A for ciklusban a , műveletet alkalmazzuk.
 * 1997. November 7.  Dévényi Károly, devenyi@inf.u-szeged.hu
 */

#include <stdio.h>

int main() {
    char cha;                          /* az ábécé kisbetűinek */
    char chA;                         /* az ábécé nagybetűinek */

    for (cha = 'a', chA = 'A'; cha <= 'z'; ++cha, ++chA) {
        printf("%c%c", cha, chA);
    }
    printf("\n");
    return 0;
}
```

Növekvő számlálásos ismétléses vezérlés

Megvalósítás C nyelven

- Szerkezeti ábrája:



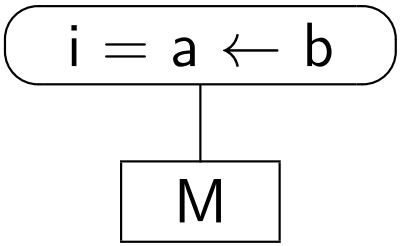
- Megvalósítása:

```
for (i = a; i <= b; ++i) {
    M;
}
```

Csökkenő számlálásos ismétléses vezérlés

Megvalósítás C nyelven

- Szerkezeti ábrája:



- Megvalósítása:

```
for (i = b; a <= i; --i) {
    M;
}
```

$\binom{n}{k}$ nemrekurzív implementáció [1/1]

P3szog.c [13–25]

```
13 int n_alatt_k(int n, int k) {
14                 /* n alatt k értékének kiszámítása nemrekurzív függvénnyel */
15     int i, nak;
16
17     if (n >= k && k >= 0) {                          /* input adatok jók-e? */
18         nak = 1;                                        /* inicializálás */
19         for (i = 1; i <= k; ++i)                              /* ciklus */
20             nak = nak * (n - i + 1) / i;
21     } else {
22         nak = 0;
23     }
24     return nak;
25 }
```

Pascal háromszög

Problémafelvetés és specifikáció

- Problémafelvetés
  - A Pascal háromszög kiszámolása és megjelenítése.

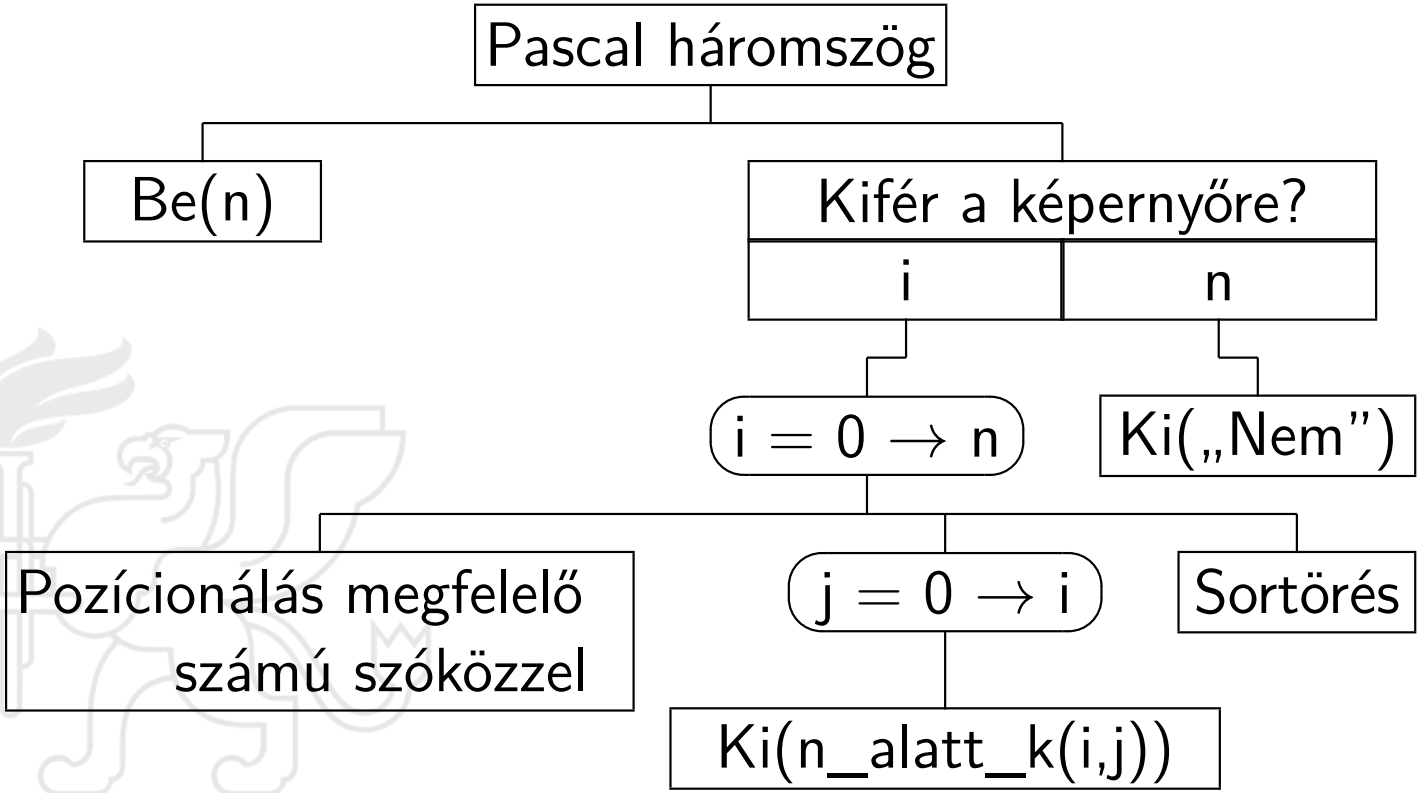
$$\begin{array}{ccccccccc}
 & & & & \binom{0}{0} & & & & \\
 & & & \binom{1}{0} & & \binom{1}{1} & & & \\
 & & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & & \\
 & \binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} & \\
\binom{4}{0} & & \binom{4}{1} & & \binom{4}{2} & & \binom{4}{3} & & \binom{4}{4}
\end{array} = \begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1
\end{array}$$

- Specifikáció
  - A probléma inputja egy n nemnegatív egész szám.
  - Az output $n+1$ darab sor piramisszerűen rendezve, rendre a Pascal háromszög sorai.

Pascal háromszög

Algoritmustervezés

- A főprogram feladata a beolvasás, input ellenőrzése, majd a nAlk függvényt alkalmas paraméterekkel aktivizálva a soronkénti kiíratás.



Pascal háromszög ciklusos $\binom{n}{k}$ implementációval [1/2]

P3szog.c [1–25]

```c
/* n alatt k értékének kiszámítása egy nemrekurzív függvénnyel és
 *   az értékek elrendezése a Pascal háromszögben.
 * 1997. Október 31.  Dévényi Károly, devenyi@inf.u-szeged.hu
 * 2013. Augusztus 29. Gergely Tamás, gertom@inf.u-szeged.hu
 * 2020. Július 30. Jász Judit, jasy@inf.u-szeged.hu
 */

#include <stdio.h>

#define SZAMSZ       5                      /* egy szám kiírási szélessége */
#define KEPSZ       80                          /* a képernyő szélessége */

int n_alatt_k(int n, int k) {
                /* n alatt k értékének kiszámítása nemrekurzív függvénnyel */
    int i, nak;

    if (n >= k && k >= 0) {                        /* input adatok jók-e? */
        nak = 1;                                       /* inicializálás */
        for (i = 1; i <= k; ++i)                               /* ciklus */
            nak = nak * (n - i + 1) / i;
    } else {
        nak = 0;
    }
    return nak;
}
```

Pascal háromszög ciklusos $\binom{n}{k}$ implementációval [2/2]

P3szog.c [27–45]

```
27 int main() {
28     int n;                                                  /* a sorok száma */
29     int i, j;                                              /* ciklusváltozók */
30
31     printf("Kérem␣a␣Pascal␣háromszög␣sorainak␣számát\n");
32     scanf("%d%*[^\n]", &n); getchar();                        /* beolvasás */
33     if (0 <= n && n <= KEPSZ / SZAMSZ - 2) {      /* kifér-e a képernyőre? */
34         for (i = 0; i <= n; ++i) {
35             printf("%*c", KEPSZ / 2 - (i + 1) * SZAMSZ / 2 - 1, '␣'); /* pozicionálás */
36             for (j = 0; j <= i; ++j) {
37                 printf("%*d", SZAMSZ, n_alatt_k(i, j));        /* kiíratás */
38             }
39             putchar('\n');
40         }
41     } else {                                                /* hibaüzenet */
42         printf("%d␣sor␣nem␣fér␣ki␣a␣␣képernyőre\n", n);
43     }
44     return 0;
45 }
```

Pascal háromszög

Problémafelvetés és specifikáció

- Problémafelvetés
  - A Pascal háromszög kiszámolása és megjelenítése.

$$\begin{array}{ccccccccc} & & & & \binom{0}{0} & & & & \\ & & & \binom{1}{0} & & \binom{1}{1} & & & \\ & & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & & \\ & \binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} & \\ \binom{4}{0} & & \binom{4}{1} & & \binom{4}{2} & & \binom{4}{3} & & \binom{4}{4} \end{array} = \begin{array}{ccccccccc} & & & & 1 & & & & \\ & & & 1 & & 1 & & & \\ & & 1 & & 2 & & 1 & & \\ & 1 & & 3 & & 3 & & 1 & \\ 1 & & 4 & & 6 & & 4 & & 1 \end{array}$$

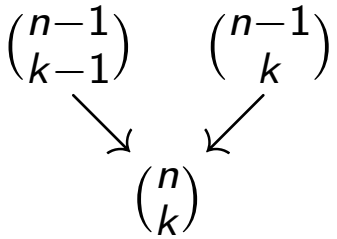
- Specifikáció
  - A probléma inputja egy n nemnegatív egész szám.
  - Az output $n+1$ darab sor piramisszerűen rendezve, rendre a Pascal háromszög sorai.

$\binom{n}{k}$ kiszámítása

Problémafelvetés és specifikáció

- Problémafelvetés
  - Számítsuk ki $\binom{n}{k}$ értékét
- Specifikáció
  - A probléma bemenete két egész szám, n és k.
  - A probléma kimenete $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ ha értelmezhető n-re és k-ra, különben 0.

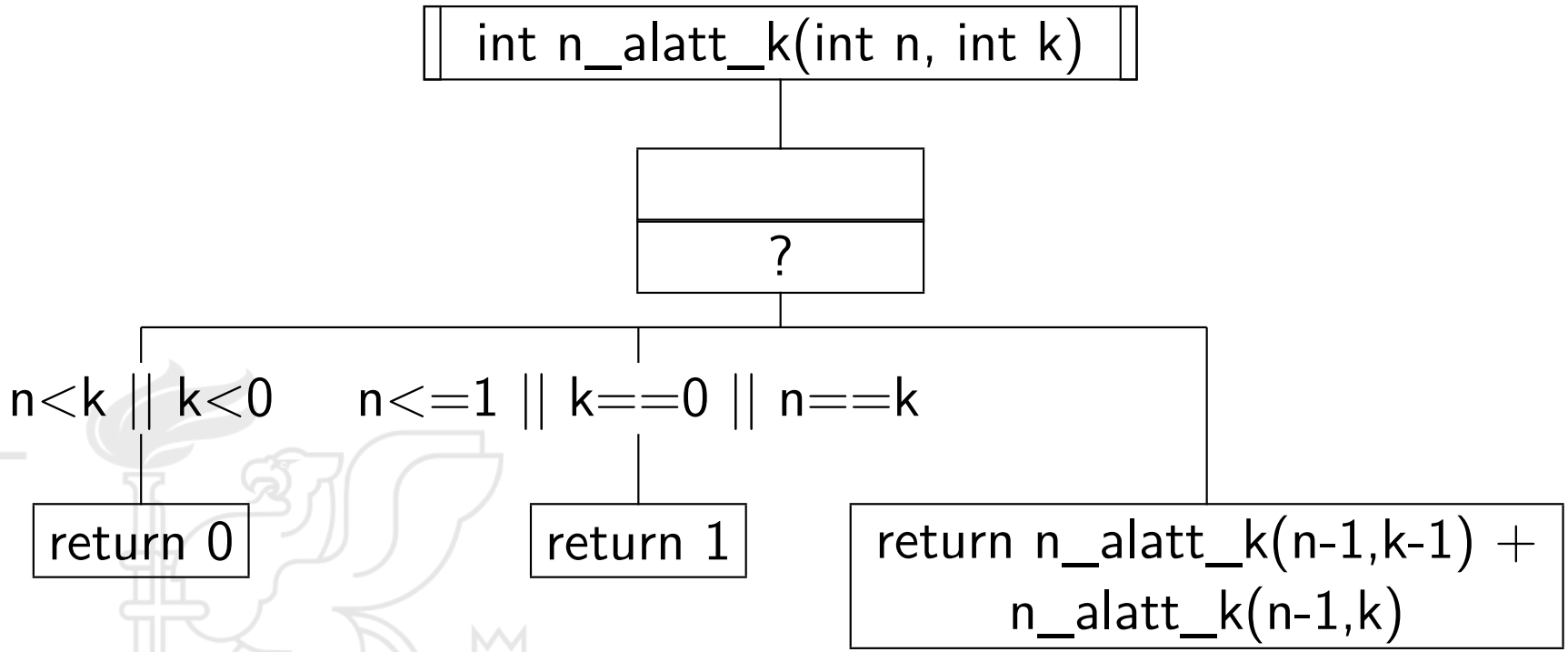
- Készítsünk az $\binom{n}{k}$ kiszámolására egy rekurzív függvényt. Ehhez felhasználjuk, hogy a Pascal háromszög szélén lévő érték 1, a belsejében lévő érték pedig a felette lévő két érték összege.



$\binom{n}{k}$ kiszámítása

Algoritmustervezés – Szerkezeti ábra

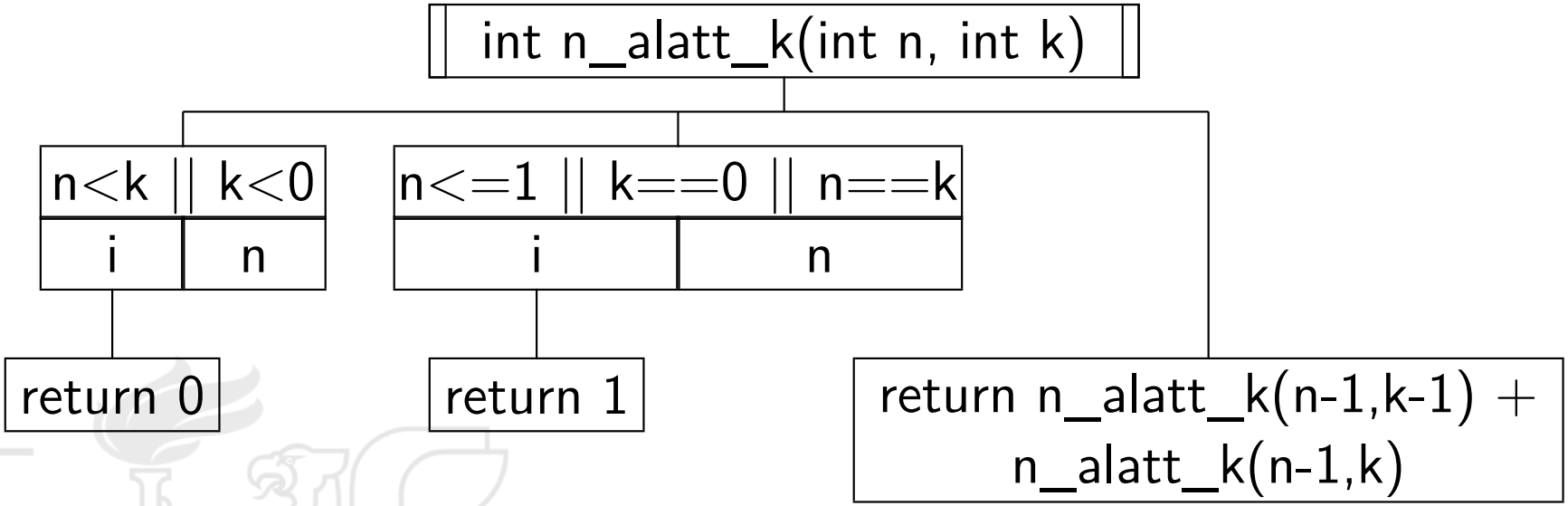
- Az $\binom{n}{k}$ értékét számító rekurzív függvény szerkezeti ábrája:



$\binom{n}{k}$ kiszámítása

Algoritmustervezés – Szerkezeti ábra (a `return` tulajdonságait figyelembe véve)

- Az $\binom{n}{k}$ értékét számító rekurzív függvény szerkezeti ábrája:



Pascal háromszög rekurzív $\binom{n}{k}$ implementációval [1/2]

P3szogr.c [1–22]

```c
/* n alatt k értékének kiszámítása egy rekurzív függvénnyel és
 *   az értékek elrendezése a Pascal háromszögben.
 * 1997. Október 31. Dévényi Károly, devenyi@inf.u-szeged.hu
 * 2015. Október 11. Gergely Tamás, gertom@inf.u-szeged.hu
 * 2020. Július 30. Jász Judit, jasy@inf.u-szeged.hu
 */

#include <stdio.h>

#define SZAMSZ       5                      /* egy szám kiírási szélessége */
#define KEPSZ       80                          /* a képernyő szélessége */

int n_alatt_k(int n, int k) {
                  /* n alatt k értékének kiszámítása rekurzív függvénnyel */
    if (n < k || k < 0) {                            /* input adatok jók-e? */
        return 0;
    }
    if (n <= 1 || n == k || k == 0) {                          /* alapesetek */
        return 1;
    }
    return n_alatt_k(n - 1, k - 1) + n_alatt_k(n - 1, k);    /* rek. hívás */
}
```

Pascal háromszög rekurzív $\binom{n}{k}$ implementációval [2/2]

P3szogr.c [24–42]

```c
int main() {
    int n;                                          /* a sorok száma */
    int i, j;                                       /* ciklusváltozók */

    printf("Kérem a Pascal háromszög sorainak számát\n");
    scanf("%d%*[^\n]", &n); getchar();              /* beolvasás */
    if (0 <= n && n <= KEPSZ / SZAMSZ - 2) {    /* kifér-e a képernyőre? */
        for (i = 0; i <= n; ++i) {
            printf("%*c", KEPSZ / 2 - (i + 1) * SZAMSZ / 2 - 1, ' '); /* pozicionálás */
            for (j = 0; j <= i; ++j) {
                printf("%*d", SZAMSZ, n_alatt_k(i, j));     /* kiíratás */
            }
            putchar('\n');
        }
    } else {                                        /* hibaüzenet */
        printf("%d sor nem fér ki a  képernyőre\n", n);
    }
    return 0;
}
```

Számsorozat legnagyobb közös osztója

Problémafelvetés és specifikáció

- Problémafelvetés
  - Számítsuk ki pozitív egész számok sorozatának legnagyobb közös osztóját!
- Specifikáció
  - A probléma inputja egy pozitív egész számokból álló sorozat, melyet a 0 zár.
  - A számsorozat elemeinek legnagyobb közös osztója.

Számsorozat legnagyobb közös osztója

Algoritmustervezés

- Az algoritmus lényege egy olyan ismétléses vezérlés, amely ciklusmagjának egyszeri végrehajtása kiszámítja a már feldolgozott input számsor és a beolvasott következő szám legnagyobb közös osztóját.
  - Ez azt jelenti, hogy minden ciklusléptésben ki kell számolnunk két szám legnagyobb közös osztóját.
- Látható, hogy a ciklusmag ismétlését célszerű a ciklusmagban vezérelni, mert az ismétlés befejeződhet úgy, hogy
  - végére értünk az inputnak, vagy
  - a feldolgozott sorozat legnagyobb közös osztója 1.

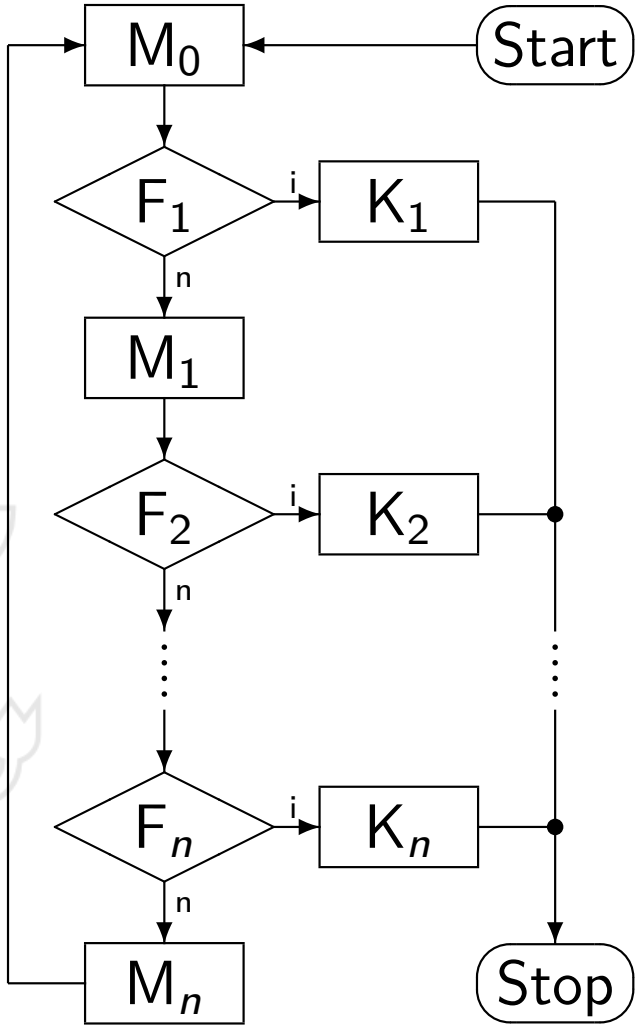
Hurok ismétléses vezérlés

- Amikor a ciklusmag ismétlését a ciklusmagon belül vezéreljük úgy, hogy a ciklus különböző pontjain adott feltételek teljesülése esetén a ciklus végrehajtását befejezzük, hurok ismétléses vezérlésről beszélünk.
- Legyenek F_i logikai kifejezések, K_i és M_j pedig tetszőleges (akár üres) műveletek $1 \leq i \leq n$ és $0 \leq j \leq n$ értékekre. Az F_i kijárati feltételekből, K_i kijárati műveletekből és az M_i műveletekből képzett hurok ismétléses vezérlés a következő előírást jelenti:
  1. Az ismétléses vezérlés következő végrehajtandó egysége az M_0 művelet.
  2. Ha a végrehajtandó egység az M_j művelet, akkor ez végrehajtódik. $j = n$ esetén folytassuk az 1.) lépéssel, különben pedig az F_{j+1} feltétel végrehajtásával a 3.) lépésben.
  3. Ha a végrehajtandó egység az F_i feltétel ($1 \leq i \leq n$), akkor értékeljük ki. Ha F_i igaz volt, akkor hajtsuk végre a K_i műveletet, és fejezzük be a vezérlést. Különben a végrehajtás az M_i művelettel folytatódik a 2.) lépésben.

Hurok ismétléses vezérlés

Folyamatábra

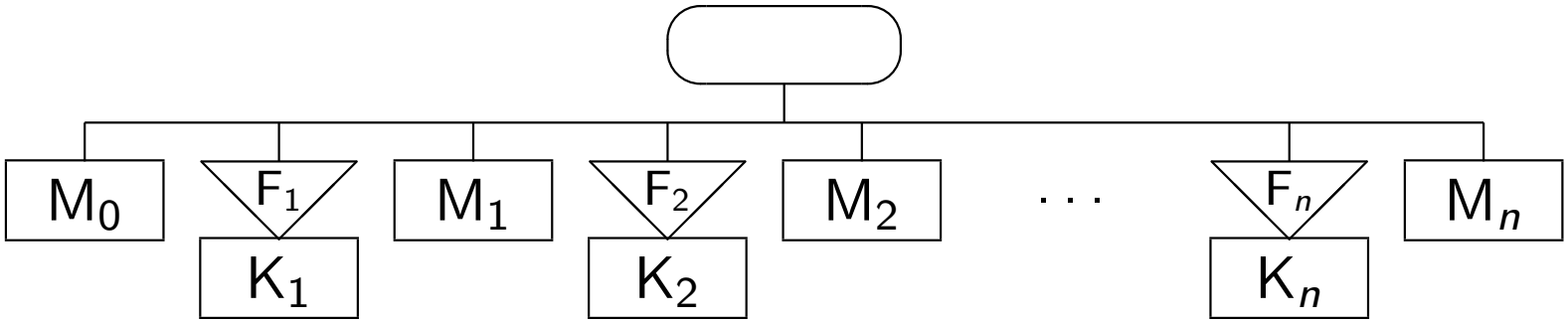
- Folyamatábrája:



Hurok ismétléses vezérlés

Szerkezeti ábra

- Szerkezeti ábrája:



Hurok ismétléses vezérlés

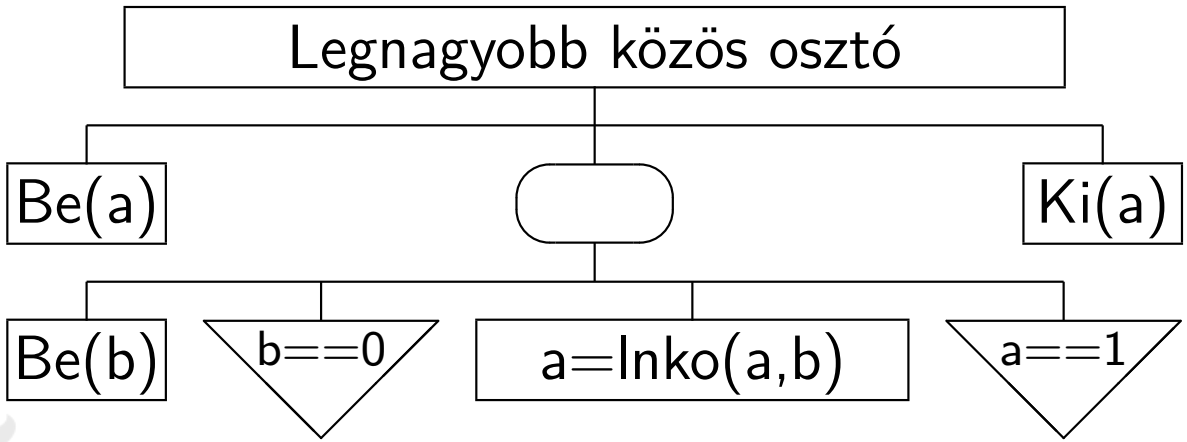
Kapcsolat más ismétléses vezérlésekkel

- A kezdő- és végfeltételes ismétléses vezérlések speciális esetei a hurok ismétléses vezérlésnek.

Számsorozat legnagyobb közös osztója

Algoritmustervezés – Szerkezeti ábra

- A főprogram:



Két szám legnagyobb közös osztója

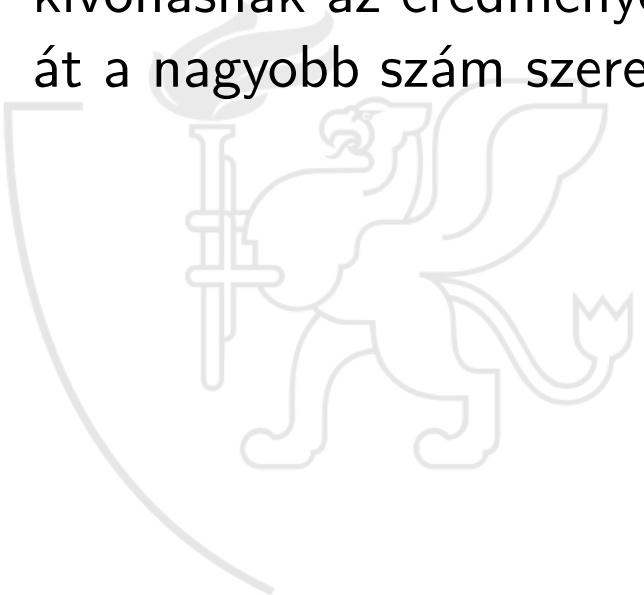
Problémafelvetés és specifikáció

- Problémafelvetés
  - Számítsuk ki két nemnegatív egész szám legnagyobb közös osztóját!
- Specifikáció
  - Készítsünk függvényt.
  - Az input x és y nemnegatív egész számok.
  - Az output egy egész szám, x és y legnagyobb közös osztója.

Két szám legnagyobb közös osztója

Algoritmustervezés

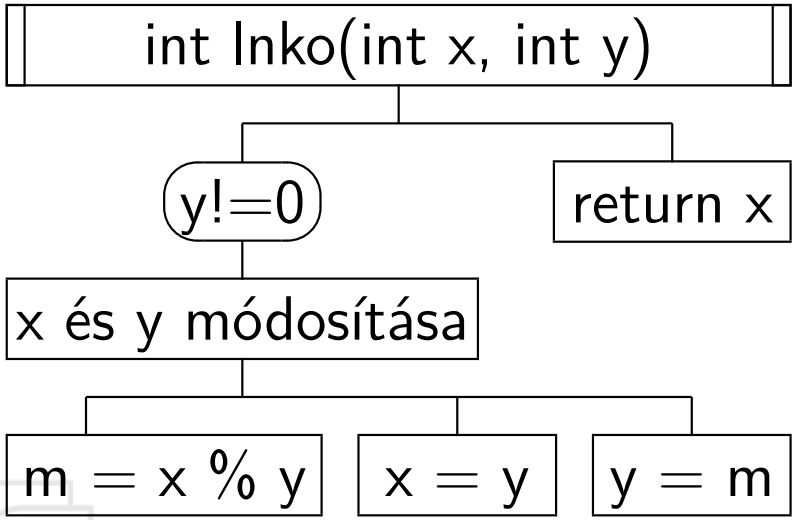
- Euklidesz algoritmusát valósítjuk meg, melynek alapja, hogy
  - ha $a \geq b > 0$ akkor $\text{lnko}(a, b) = \text{lnko}(a - b, b)$, és
  - $\text{lnko}(a, 0) = a$, vagyis a nagyobbik számból a kisebbiket kivonogatva előbb-utóbb megkapjuk a két eredeti szám legnagyobb közös osztóját.
- A gyakorlatban az ismételt kivonogatás helyett maradékos osztást alkalmazunk, ami egyetlen lépésben megadja annak az $a' - b$ kivonásnak az eredményét, amikor $a' - b < b$ teljesül, vagyis b veszi át a nagyobb szám szerepét.



Két szám legnagyobb közös osztója

Algoritmustervezés – Szerkezeti ábra

- A legnagyobb közös osztó függvény:



Két szám legnagyobb közös osztója [1/1]

lnkoszt.c [9–22]

```
 9 int lnko(int x, int y) {
10     /* x és y legnagyobb közös osztójának meghatározása
11      * Euklidesz algoritmusával.
12      */
13     int m;
14
15     while (y != 0) {
16         m = x % y;
17         x = y;
18         y = m;
19     }
20     return x;
21 }
```

Hurok vezérlés

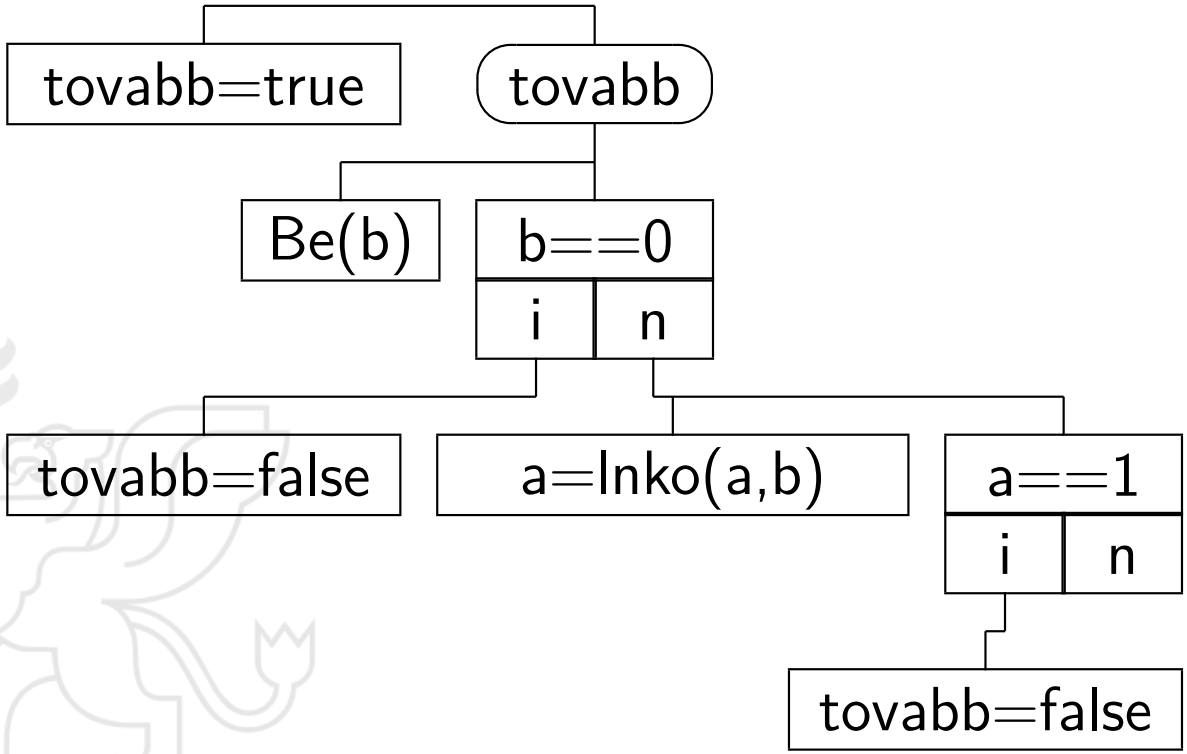
Megvalósítás #1

- A C nyelvben nincs olyan vezérlési forma, amellyel közvetlenül megvalósíthatnánk a hurok ismétléses vezérlést, de a kezdőfeltételes ismétléses vezérlés és egyszerű szelekció segítségével kifejezhetjük. Ez a megvalósítás nyelvfüggetlen.
- A megvalósítás lényege, hogy választunk egy logikai változót (legyen az a `tovabb`), ez lesz az ismétlés feltétele. Továbbá a ciklusmagban a feltételes kijáratokat úgy alakítjuk át, hogy
  - ha a feltétel igaz, akkor a hozzá tartozó kijárati művelet végrehajtásra kerül és a `tovabb` hamis értéket kap,
  - egyébként végrehajtódik a ciklusmag további része.

Hurok vezérlés

Megvalósítás #1

- A LNKOszt algoritmusban szereplő hurok ciklus tehát a következőképpen valósítható meg:



Számsorozat legnagyobb közös osztója$_1$ [1/2]

lnkoszt1.c [1–21]

```c
/* Pozitív egész számok legnagyobb közös osztójának meghatározása.
 *   A hurok ismétléses vezérlés megvalósítása ismert szerkezetekkel.
 * 1997. November 7.  Dévényi Károly, devenyi@inf.u-szeged.hu
 * 2006. Augusztus 8.  Gergely Tamás, gertom@inf.u-szeged.hu
 */

#include <stdio.h>

int lnko(int x, int y) {
    /* x és y legnagyobb közös osztójának meghatározása
     * Euklidesz algoritmusával.
     */
    int m;

    while (y != 0) {
        m = x % y;
        x = y;
        y = m;
    }
    return x;
}
```

Számsorozat legnagyobb közös osztója$_1$ [2/2]

lnkoszt1.c [23–49]

```
23 int main() {
24     int a, b;
25     int tovabb;                  /* logikai változó a ciklus megvalósításához */
26
27     printf("A program pozitív egész számok legnagyobb\n");
28     printf("közös osztóját számítja.\n");
29     printf("Kérem a számok sorozatát, amit 0 zár!\n");
30     printf("? ");
31     scanf("%d%*[^\n]", &a); getchar();
32
33     tovabb = !0;
34     while (tovabb) {                                  /*  a hurok ciklus kezdete */
35         printf("? ");
36         scanf("%d%*[^\n]", &b); getchar();
37
38         if (b == 0) {                                          /* első kijárat */
39             tovabb = 0;
40         } else {
41             a = lnko(a, b);
42             if (a == 1) {                                  /* második kijárat */
43                 tovabb = 0;
44             }
45         }
46     }                                                  /* a hurok ciklus vége */
47     printf(" A számok legnagyobb közös osztója: %d\n", a);
48     return 0;
49 }
```

A `break` és `continue` utasítások

- A C nyelvben a ciklusmag folyamatos végrehajtásának megszakítására két utasítás használható:

`break` Megszakítja a ciklust, a program végrehajtása a ciklusmag utáni első utasítással folytatódik. Használható a `switch` utasításban is, hatására a program végrehajtása a `switch` utáni első utasítással folytatódik.

`continue` Megszakítja a ciklusmag aktuális lefutását, a vezérlés a ciklus feltételének kiértékelésével (`while`, `do while`) illetve az inkrementáló kifejezés kiértékelésével (`for`) folytatódik.

Hurok vezérlés

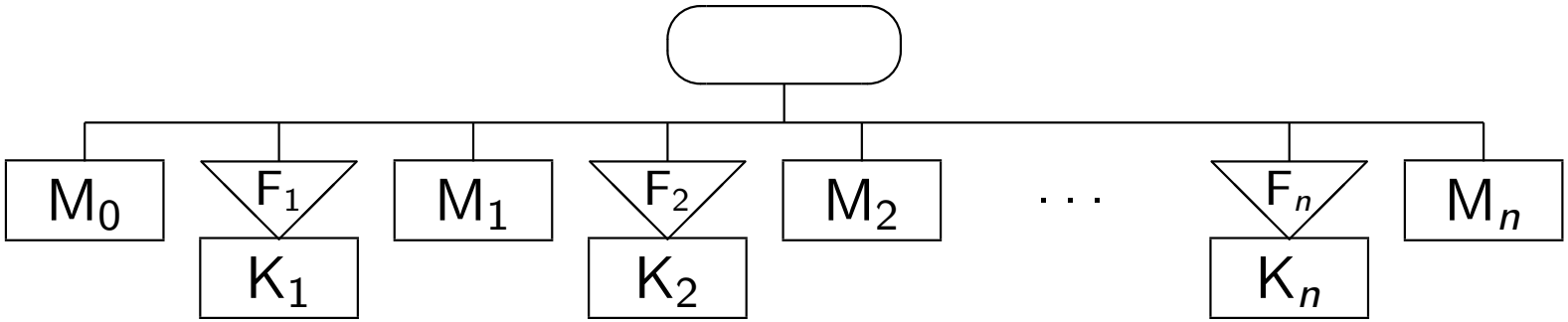
Megvalósítás #2

- A hurok ismétléses vezérlés második megvalósítása a C nyelv `break` utasítását használja.
- A `break` utasítás „megtöri” az aktuális ismétléses (vagy mint láttuk, esetkiválasztásos szelekciós) vezérlést, és a vezérlési szerkezet utáni első utasításnál folytatja a programot.

Hurok vezérlés

Megvalósítás C nyelven

- A hurok ismétléses vezérlés szerkezeti ábrája:



- Megvalósítása:

```
while (1) {
    M0;
    if (F1) {
        K1; break;
    }
    M1;
    ...
```

```
    ...

    if (Fn) {
        Kn; break;
    }
    Mn;
}
```

Számsorozat legnagyobb közös osztója [1/2]

lnkoszt.c [1–21]

```c
/* Pozitív egész számok legnagyobb közös osztójának meghatározása.
 *   A hurok ismétléses vezérlés megvalósítása break utasítással.
 * 1997. November 7.  Dévényi Károly, devenyi@inf.u-szeged.hu
 * 2006. Augusztus 8.  Gergely Tamás, gertom@inf.u-szeged.hu
 */

#include <stdio.h>

int lnko(int x, int y) {
    /* x és y legnagyobb közös osztójának meghatározása
     * Euklidesz algoritmusával.
     */
    int m;

    while (y != 0) {
        m = x % y;
        x = y;
        y = m;
    }
    return x;
}
```

Számsorozat legnagyobb közös osztója [2/2]

lnkoszt.c [23–45]

```c
int main() {
    int a, b;

    printf("A program pozitív egész számok legnagyobb\n");
    printf("közös osztóját számítja.\n");
    printf("Kérem a számok sorozatát, amit 0 zár!\n");
    printf("? ");
    scanf("%d%*[^\n]", &a); getchar();

    while (1) {                                    /* a hurok ciklus kezdete */
        printf("? ");
        scanf("%d%*[^\n]", &b); getchar();
        if (b == 0) {                                        /* első kijárat */
            break;
        }
        a = lnko(a, b);
        if (a == 1) {                                     /* második kijárat */
            break;
        }
    }                                                  /* a hurok ciklus vége */
    printf(" A számok legnagyobb közös osztója: %d\n", a);
    return 0;
}
```

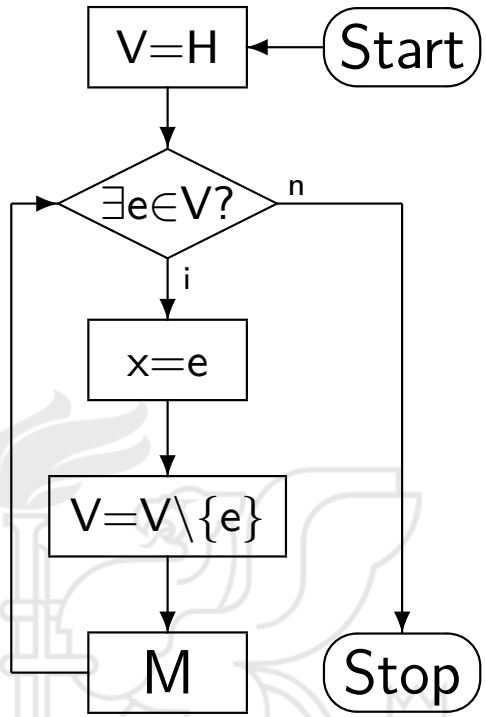
Diszkrét ismétléses vezérlés

- Diszkrét ismétléses vezérlésről akkor beszélünk, ha a ciklusmagot végre kell hajtani egy halmaz minden elemére tetszőleges sorrendben.
- Legyen x egy T típusú változó, H a T értékkészletének részhalmaza, M pedig tetszőleges művelet, amelynek nincs hatása x és H értékére. A H halmazból, x ciklusváltozóból és M műveletből (ciklusmagból) képzett diszkrét ismétléses vezérlés az alábbi vezérlési előírást jelenti:
  1. Ha a H halmaz minden elemére végrehajtottuk az M műveletet, akkor vége a vezérlésnek.
  2. Egyébként vegyük a H halmaz egy olyan tetszőleges e elemét, amelyre még nem hajtottuk végre az M műveletet, és folytassuk a 3.) lépéssel.
  3. Legyen $x = e$ és hajtsuk végre az M műveletet, majd folytassuk az 1.) lépéssel.

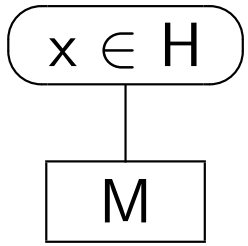
Diszkrét ismétléses vezérlés

Folyamatábra és szerkezeti ábra

- Folyamatábrája:



- Szerkezeti ábrája:



- A *H* halmaz számossága határozza meg, hogy az *M* művelet hányszor hajtódik végre. Ha a *H* az üres halmaz, akkor a diszkrét ismétléses vezérlés az *M* művelet végrehajtása nélkül befejeződik.

Diszkrét ismétléses vezérlés

Megvalósítás C nyelven

- A diszkrét ismétléses vezérlésnek nincs közvetlen megvalósítása a C nyelvben.
- A megvalósítás elsősorban attól függ, hogy az ismétlési feltételben megadott halmazt hogyan reprezentáljuk.
- Algoritmustervezés során szabadon használhatjuk a diszkrét ismétléses vezérlést, ha erre van szükség a probléma megoldásához. A halmaz reprezentálásáról pedig akkor döntünk, amikor elegendő információ áll rendelkezésünkre, hogy a legmegfelelőbbet kiválaszthassuk.