Prog 1 összefoglaló:

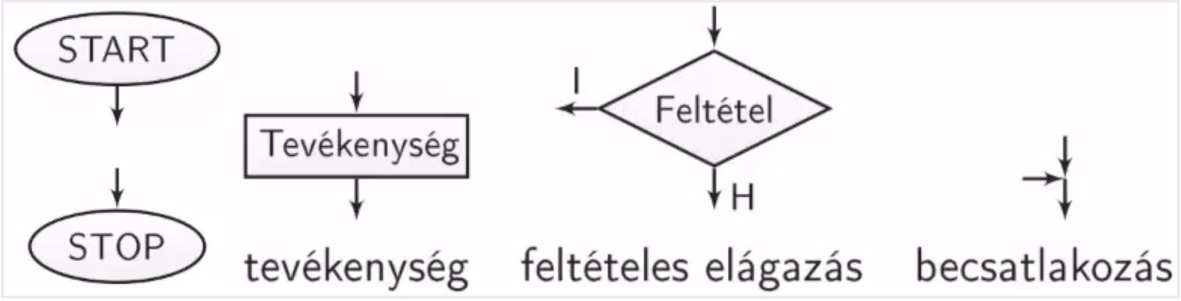
https://en.cppreference.com/w/

God himself: https://www.youtube.com/watch?v=pdfLW9PSrhQ

Run: Crtl+Alt+N
Stop: Ctrl+Alt+M

Az imperatív programozási paradigma -- adatok, típusok, kifejezések:

- Imperatív programozás: algoritmust (módszert) találunk ki majd ezt lekódoljuk.
- Algoritmus (módszer): Gépiesen végrehajtható lépések véges sorozata, amely elvezet a megoldáshoz.
- Egy algoritmus:
  - Helyes, ha tényleg azt oldja meg amit szeretnénk
  - Teljes, ha minden lehetséges esetben megoldja
  - Véges, ha véges sok lépésben befejeződik.
- Az algoritmusok nyelvfüggetlen leírási módjai:
  - Pszeudokód (álkód): Természetes nyelven, de precízen megfogalmazott utasítássorozat.
  - Grafikus ábrázolás (folyamatábra):



https://www.tutorialspoint.com/format-specifiers-in-c

- Típusok:
  - ***Szám***:
    - Értéke: 0, -1, e, π
    - Műveletek: összeadás, kivonás, összehasonlítás, rendezés
    - Kiiratási kódja: %d
  - ***Betű***:
    - Értéke: a, A, b
    - Műveletek: összehasonlítás, rendezés
  - ***Logikai***:

        - Értéke: {igaz, hamis}
        - Műveletek: tagadás, konjukció (ÉS), diszjunkció (VAGY)
    - ***Szín***:
        - Értéke: piros, kék, …
        - Műveletek: összehasonlítás
    - ***Hőmérséklet***:
        - Értéke: hideg, meleg, forró
        - Műveletek: összehasonlítás, rendezés
- Az alapján hogy az algoritmusban a változó milyen szerepet tölt be lehet:
    - Állandó (konstans): értéke nem változik az algoritmus futtatása során és a típusa a megjelenési formából kiderül.
    - Változó:
        - Azonosítóval jelöljük, értéke műveletekben felhasználható (olvasás), értéke frissíthető (értékadás, írás), a típusát mindig külön kell megadni (deklaráció).
- Kifejezés: Állandókból és változókból a megfelelő műveletek (operációk) alkalmazásával kifejezések képezhetők.
    - Kiértékelhető típusa és értéke van, a műveleteket operátorok határozzák meg, melyek az operandusokon dolgoznak.
    - A kifejezés típusa nem feltétlen egyezik meg az operandusok (összetevők) típusával.
- Szintaxis: Szintaktika (nyelvtani) hiba (syntax error)
    - Helytelenül használtuk a nyelv szabályait, így a program értelmezhetetlen, ez általában hamar kiderül, egyszerűen, gyorsan javítható
- Szemantikai (értelmezési) hiba (semantic error):
    - A program végrehajtható, lefut, de nem azt csinálja amit akarunk, ez sokszor nehezen megtalálható, reprodukálható, nehéz javítani.
- Kommentek: **//** vagy **/* */**

Strukturált programok - C vezérlési szerkezetek:

Struktúrák:

- Struktúra: logikailag egységet alkotó, akár különböző típusú adatokból álló, összetett adattípus.
  - A részadatokat mezőknek vagy tagoknak hívjuk
  - Egyetlen értékadással másolhatók
  - Lehet függvény paramétere és visszatérési értéke is

Pl:

```
struct fasz {double h; double a;};

struct fasz(struct fasz a, struct fasz b) {
    /* kód */
}

int main() {
    struct fasz f1, f2 = {11.1, 3.5}; //változó
definiálása és értékadás
    f1.h = 21.2; //egyenkénti érték adások
    f1.a = 1.1;
}
```

Típusnév-hozzárendelés:

- C-ben a "***typedef***" kulcsszóval csinálhatunk álneveket (alias-t) egy típushoz, ezután a típus az eredeti és az álnevével is elérhető lesz.
- A fő célja hogy beszédesebbé tegyük a kódunkat.
- Ezt egyesíthetjük egy struktúrával is, az alábbi módon:

```
typedef struct fasz {double hossz; double
```

```
atmero;} fasz;
```

- Mivel a typedef -fel adunk a struktúránknak egy álnevet így a struktúra neve elhanyagolhatóvá válik:

```
typedef struct {double hossz; double atmero;}
fasz;
```

Indirekció:

- Memória cím kiiratása: "***printf("%p", &név)***"
  - Ide azért kell az **&** jel mert ez a címképzés operátor
- Memória címek tárolására való mutató (pointer) definíciója:

```
<mutatott típus> * <azonosító>
```

Pl:

```
int*    p; /* p egy int adat címét tárolja    */
double* q; /* q egy double adat címét tárolja */
char*   r; /* r egy char adat címét tárolja   */
```

vagy

```
int    *p; /* p egy int adat címét tárolja    */
double *q; /* q egy double adat címét tárolja */
char   *r; /* r egy char adat címét tárolja   */
```

Az índirekció művelete:

- Ha **a** p mutat az a változó címét tartalmazza, akkor **p** "**a**-ra mutat"
- Ha **p** a -ra mutat, akkor az a változó ***p** -ként elérhető. Itt * az índirekció operátora (dereferencia operátor)

Pl:

```
int a, b;
int *p; /* int pointer */

a = 2;
b = 3;
p = &a; /* p a-ra mutat */
*p = 4; /* a = 4 */
p = &b; /* p b-re mutat */
*p = 5; /* b = 5 */
```

- Címképzés és indirekció - összefoglalás:

```
operátor    művelet      leírás
-------------------------------
&           címképzés    változóhoz a címet
rendeli
*           indirekció   címhez a változót
rendeli
```

- Csak is arra a típusú változóra tud mutatni amilyen típussal definiálva lett.
- Az índirekció egyik felhasználási módja pl: globális változók megváltoztatása egy függvényen belülről, ezt úgy is mondjuk hogy a változókat cím szerint adjuk oda a függvénynek, így a változók a függvény számára módosíthatók lesznek.

Mutatók és tömbök:

- Mutató-aritmetika:
  - Ha **p** és **q** azonos típusú mutató, akkor:

```
kif.      típus         érték
-----------------------------
p+1       mutató        a következő elem címe
p-1       mutató        az előző elem címe
q-p       egész szám    két cím közötti elemek
szzáma
```

Pl:

```
int a, *p, *q;

p = &a;
p = p-1;
q = p+2;
printf("%d", q-p);
```

```
Output: 2
```

- Mutatóaritmetikai műveleteknél a címeket nem bájtban, hanem a mutatott típus ábrázolási méretében mérjük
- Ha egy mutatót ráállítok egy tömbre akkor annak a tömbnek az elemeit be tudom járni a mutató segítségével
- A mutatók indexelhetőek:

```
int main() {
    int t[4] = {2, -5, 4, 6};
    int i;
    int *p;
    p = &t[0]; /* itt átadjuk a pointernek a t
tömb első (0. dik) elemének címét, ezt p = t;
módon is megadhatjuk */
    for (i = 0; i < 4; i++) {
        printf("%d", p[i] /* *(p+i) */); /*
mivel jelenleg a p pointer tartalmazza a t tömb
első címét így ugyan úgy végig tudunk rajta
iterálni mint egy normális tömbön és akár így is
jelölhetjük p[i] */
    }
}
```

- Mutató és tömb közötti különbség: Ha egy tömböt hozok létre akkor a tömb elemeinek foglalok memóriát, ha pedig egy mutatót akkor csak egy címnek.
- C -ben Tömböt függvénynek mindig úgy adunk át hogy megadjuk a tömb méretét.
- Tömb függvénynek való átadása:

```
int /* ha oda rakjuk a csillag operátort akkor a
függvény a talált elemenek a címét fogja
visszaadni */ *elso_negativ(int *r /* r[] -ként
is lehet jelölni*/, int meret) { /* megadjuk
hogy egy mutatott címet és a tömb méretét kérjük
*/
    for (int i = 0; i < meret; i++) {
        if (r[i] < 0)
            return &r[i]; /* r + i -vel is
jelölhetjük */
```

```
    return NULL; /* a NULL mutató egy speciális
0 értékű mutató amit arra használunk hogy
jelezzük hogy nincs megfelelő visszatérési érték
*/
}

int main() {
    int t[4] = {2, -5, 4, 6};
    int *px = elso_negativ(t, 4): /* a függvény
meghívásakor a tömb kezdőcímét adom át neki */
    if (px == NULL)
        printf("nincs negativ elem");
    else
        printf("az elso negativ: %d", *px);
    return 0;
}
```

Stringek:

- C -ben a stringeket chat tipusú tömbökben lehet tárolni
- Definiálni kell egy tömböt aminek nem szükséges megadni a méretét amennyiben azonnal megadjuk az elemeit, ha ezeket karakterenként adjuk meg akkor az utolsó karakternek ‘\0’ -nak kell lennie, ezzel jelezvén hogy amit megadtunk az egy string, emellett egyszerűen egybe írt szövegként is megadhatjuk a stringet amivel automatikusan utánna fog kerülni a ‘\0’ is.
- Tömb méretébe minden esetben bele lesz számolva a lezáró ‘\0’ karakter is.
- Ettől függetlenül a tömb méretét mi is megadhatjuk manuálisan és nagyobb is lehet mint amennyi karaktert tartalmaz a deklarációkor.
- Tömbből tömbbe nem lehet csak úgy könyvtár használata nélkül másolni.
- Tömb másolása:

```
void my_strcpy(char *to, char *from) {
    for (int i = 0; from[i] != '\0'; i++) {
        to[i] = from[i];
```

```
    }
    to[i] = '\0';
}
```

- Tömb össze hasonlítása:
  - Ha az első > második az eredmény pozitív
  - Ha az első < második az eredmény negatív
  - Ha a kettő megegyezik akkor az eredmény 0
- Ezzel nem csak egy igaz hamis értéket adunk vissza hanem rendezési információt is
- Azt a stringet tekintjük nagyobbnak amelyik hátrébb helyezkedik el a névsorban, pl: bela > aladar

```
int my_strcmp(char *s1, char *s2) {
    for(int i = 0; s1[i] != '\0')
        if(s1[i] != s2[i])
            return s1[i] - s2[i];
    return s1[i] - s2[i];
}
```

- Ezeket a függvényeket a <string.h> könyvtár importálásával is tudjuk használni (strcpy, strcmp).

Stringek beolvasása és kiiratása:

- Ha stringed akarunk kiiratni printf függvénnyel akkor a %s karakter formátum kódot kell használni

```
char str[10] = "ez szoveg";
printf("%s", str):
```

- A beolvasásnál pedig szintén a **%s** forma kódot kell használni, de ez csak az első white space karakterig olvas majd be

```
scanf("%d", &str);
```

- Ha azt akarjuk hogy mindaddig olvasson amíg nem nyomunk entert a következőt kell írjuk:

```
scanf("%s[^\n]", str);
```

- Ha limitált mennyiségű karaktert tudunk csak beolvasni akkor

az

```
fgets(tömb, max méret, honnan(stdin));

pl:

fgets(str, 10, stdin);
```

függvénnyel érdemes megtenni azt.

```
char *p = "jo"; /* ha így definiálunk string
tömböt az read only */


<string.h> függvények:

strcpy(hova, honnan); /* másolás */

strcat(hova, mit) /* stringek egymás mögé fűzése
*/

strlen(string(tömb)) /* a tömbben tárolt string
hossza, csak az értelmes karakterek hosszát adja
vissza */

strche(hol, mit) /* egy stringben keresi meg egy
karakter első előfordulását, ez egy mutatót ad
vissza, ha talál valamit akkor a karakter címét
adja vissza, ha nem akkor egy NULL mutatót */
```

Dinamikus memóriakezelés:

- Akkor kell dinamikus memóriakezelést használni ha nem tudjuk a program írásakor, hogy pontosan mekkora méretű lesz egy tömbünk.
- Ilyenkor a program futása közben mondjuk meg az operációs rendszernek hogy épp mennyi memóriára lesz szükségünk.
- Ehhez kelleni fog az ***<stdlib.h>*** könyvtár

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

Memória foglalás:

- Memória foglalásra használt függvény a "***malloc()***" .
- Ha meg szeretnénk tudni hogy mekkora egy konkrét változó típus mérete byte-okban akkor azt a "***sizeof()***" függvénye tehetjük meg. Ez mondjuk akkor lehet hasznos ha memóriát akarunk foglalni és meg akarjuk tudni hogy hányszor hány byte-ot foglaljunk.
- Amennyiben sikerült lefoglalni a memória helyet egy a foglalt memória kezdőcímét fogja vissza adni. Ez egy void * tipusú pointer lesz, ez egy típus nélküli pointer, voltaképpen csak egy cím, ami mellé típus információ nincs társítva. Ennek az az előnye van hogy könnyen átkonvertálhatjuk egy definiált típusú pointerré.

Pl:

```
double *p = (double *)malloc(n *
sizeof(double));
```

- A felül látható ***(double *)*** kifejezés egy kényszerített típus konverzió, ilyenkor nem double hanem double * típust kap, tehát pointer típusú lesz.
- Ha a "***malloc()"*** függvény nem tud lefoglalni ennyi memóriát akkor a függvény visszatérési ideje egy ***NULL*** pointer lesz és ezt mindig ellenőriznünk kell és amennyiben nincs rendelkezésre álló memória ki kell lépni a programból mivel az onnantól nem lesz funkcionális.

```
if(p == NULL)
   return 1;

// ez után jöhet a kód
```

Memória visszaadása:

- Ha befejeztük a munkát ezzel a tömbbel, akkor a lefoglalt memóriát vissza kell adjuk az operációs rendszernek.
- Ezt az "***<stdlib.h>***" könyvtár "***free()***" függvényével tehetjük meg, ami annak a pointernek a nevét kéri aminek memóriát foglaltunk, visszatérési értéke nincsen.

- Ez után a pointer még mindig oda fog mutatni, de már nem szabad indexelni (dereferálni).
- Ilyenkor nem kell azzal törődni hogy mekkora a lefoglalt memória mérete, az operációs rendszer automatikusan vissza fogja venni az egészet amire az adott pointer mutat.
- Nagyon fontos, hogy mindig visszaadjuk a memóriát, mivel amennyiben ezt nem tesszük meg a programunk el fogja kezdeni elfogyasztani a rendszer összes memóriáját egészen addig amíg csak tudja.

- Azt hogy pontosan mennyi memóriát foglaltunk le utólag nem lehet lekérdezni, mivel vagy pontosan annyit amennyit kértünk vagy ***NULL*** -t, ebből következik hogy a program futása alatt pontosan ugyan annyiszor kell "***malloc()***" -ot végrehajtani mint "***free()***" -t.

Másik függvény memória foglalásra:

- Ennek "***calloc()***" a neve (cleared alloc), ez a "***malloc()***" -al szemben két paramétert kér, azt hogy hány elemet szeretnénk foglalni és hogy egy elem mérete mekkora lesz.
- Annyiban különbözik még a "***malloc()***" függvénytől hogy a "***calloc()***" a lefoglalt memóriát ki is nullázza, tehát a tömbünk biztosan csak nulla értékeket fog tartalmazni.

Pl:

```
int n;
scanf("%d", &n);
double *p = (double *)calloc(n, sizeof(double));
```

Még egy memória foglaló függvény:

- Ha esetleg út közben kiderül hogy több vagy kevesebb memória kell mint amennyit eredetileg lefoglaltunk akkor a "***realloc()***" függvény segítségével ezt meg lehet tenni.
- A függvény két paramétert kér, az első az a cím lesz aminél a memória mennyiséget meg kell változtatni, a második pedig az új mérete kell hogy legyen.

Pl:

```
n = n + 2;
realloc(p, n * sizeof(double));
```

- Ha ezt a függvényt akkor a végrehajtásnál két dolog történhet:
  - Ha az operációs rendszer talál szabad memóriát az az előtt lefoglalt hely után akkor legfoglalja az utánna lévő helyet,
  - ha viszont nincsen közvetlen utánna hely a memóriában, akkor az operációs rendszer keres egy megfelelő méretű helyet oda átmásolja a már meglévő foglalt memóriában lévő adatokat és a maradék frissen hozzáadott helyet üresen hagyja.
- Mi pontosan egyik esetben se tudjuk hogy melyik történik pont ezért a "***realloc()***" függvénytől az új pointert (visszatérési értéket) mindenképpen el kell kérjük.

```
p = realloc(p, n * sizeof(double));
```

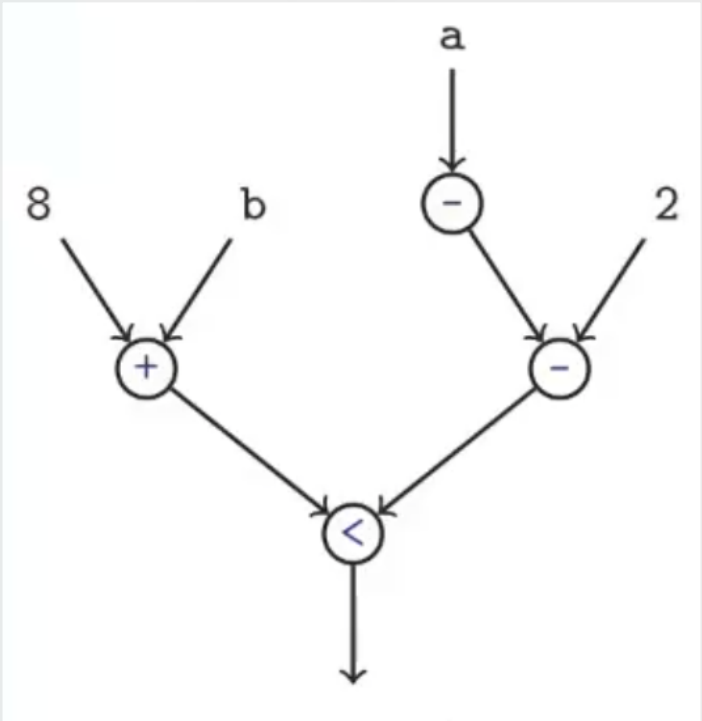
- Ez lehet, hogy pontosan az lesz mint az eredeti pointer, de az is lehet hogy kapunk egy teljesen újat, de ez nekünk mind a két esetben lényegtelen.

Operátorok és kifejezések:

- A C nyelvnek kb. 50 operátora van, amelyek a kifejezések (kiértékelhető, valamilyen eredményt szolgáltató műveletek) legfontosabb építőkövei.
- Az operátorokat műveletei jelekkel jelöljük Pl.: **+**, **-**, *****, **&**, ...
- Ezeknek vannak bemeneteik (operandusoknak vagy argumentuknak nevezzük őket) és amikor ezt kiértékeljük létrejön értékként egy típusos adat.
- Emellett az operandusok többalakúak, tehát eltérő típusú operadusokra eltérő működést ad.
- Ha van egy összetett kifejezés azt a legegyszerűbben egy kifejezésfában ábrázolhatjuk.

Pl:

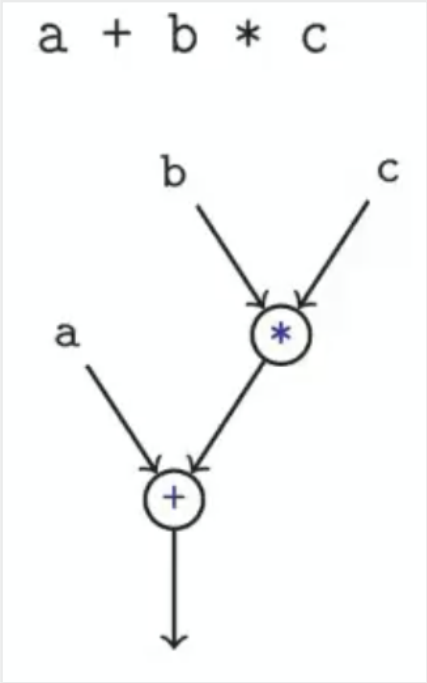
```
Kifejezés: 8 + b < -a - 2

Kifejezésfa:
```



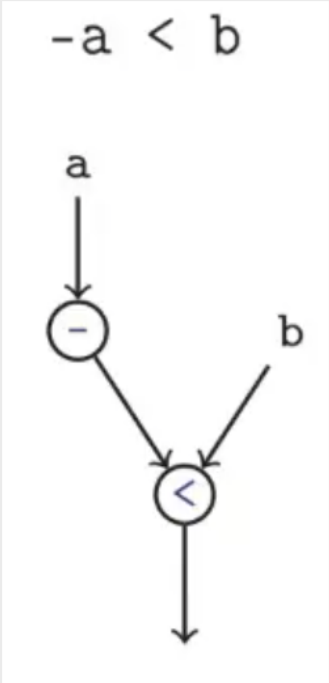
- Itt a levelek a konstansokat és változóhivatkozásokat jelképezik, a csomópontok az operátorok, a gyökér pedig a kifejezés értéke.
- Azt, hogy egy kifejezésből hogyan építhetjük fel annak a kifejezésfáját, az operátorok nyelvtana határozza meg:
  - **Precedencia:**
    - Ez akkor fontos ha két operátor találkozik, ahol az erősebb precedenciájú (matematikai erősségű) operátor értéke lesz, a gyengébb precedenciájú (matematikai erősségű) operátor argumentuma.

Pl:

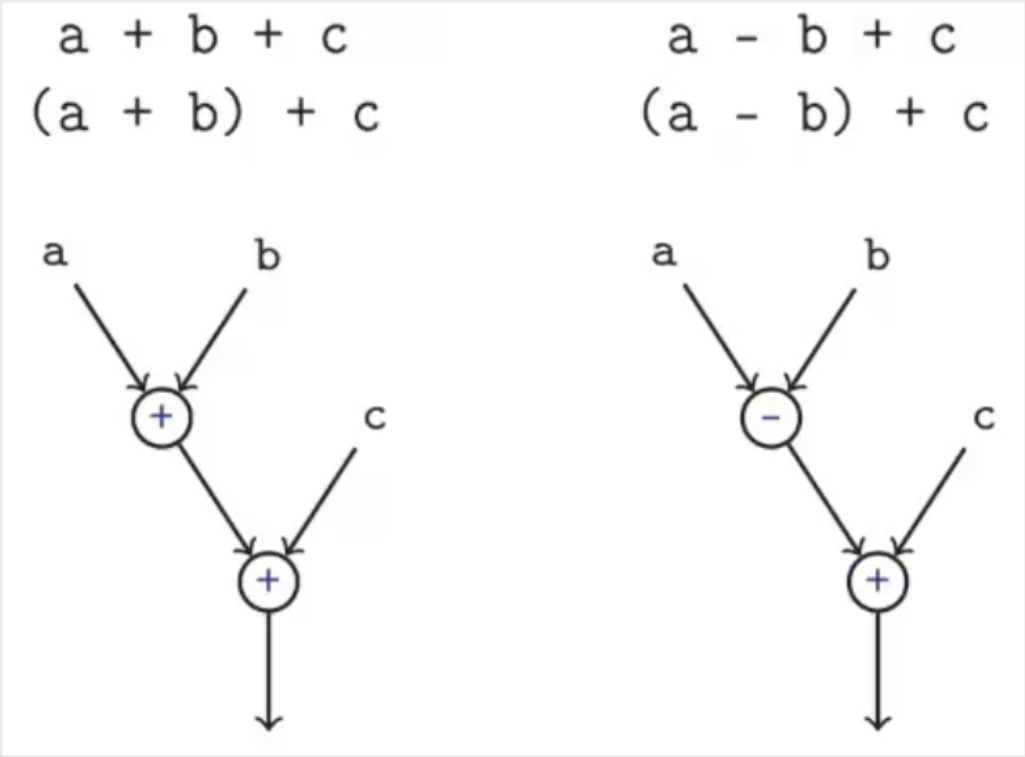
Itt a szorzás erősebb így annak az értéke lesz az összeadás egyik argumentuma



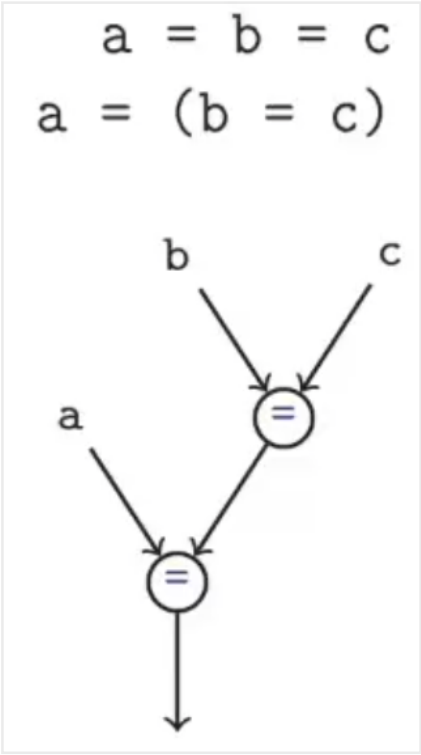
Pl:
Itt az ellentett képzés operátor gyengébb mint a kisebb operátor



- **Asszociativitás (csoportosítás):**
  - Két azonos precedenciájú operátornál van jelentősége,
    - Ha balról jobbra csoportosítunk, akkor a bal oldali operátor értéke lesz a jobb oldali operátor argumentuma, a korábbi matematikai ismereteinkben eddig mindig ilyen operátorokkal találkoztunk.



- Ha jobbról balra csoportosítunk, a jobboldali operátor értéke lesz a bal oldali operátor argumentuma. Ilyen például a programozásban az értékadás operátor.



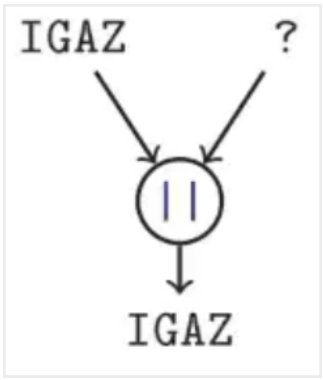
- **<u>Kifejezések kiértékelése:</u>**
  - A legtöbb operátor esetében a kiértékelés sorrendje

definiálatlan (a hatékonyság kedvéért), néhány specifikus esetben balról jobbra (a hatékonyság kedvéért).

- **A logikai rövidzár:**
  - Az || (diszjunkció) és && (konjunkció) operátorok operandusaikat balról jobbra értékelik ki, de a jobb oldalit csak akkor, ha a teljes kifejezés értéke még nem derült ki.

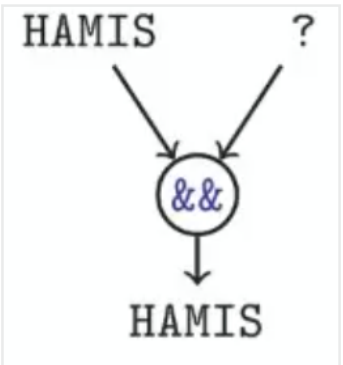
Pl:

- Például a logikai || (diszjunkció) operátornál ha a bal oldali feltétel igaz akkor a jobb oldali feltételt már ki sem értékeli.




```
char *str = "Horvath Miklos";
if (i == 0 || str[i - 1] == ' ')
  /* szó eleje */
```

- Vagy a logikai && (konjunkció) operátornál, ha a bal oldali feltétel hamis már nem fog tovább menni a jobb oldalira.

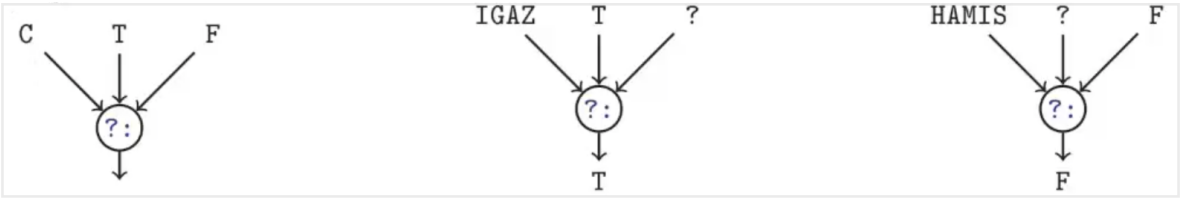



```
if (p != NULL && *p < 2)
```

```
/* ... */
```

- **A ternáris (3-operandusú) operátor:**
  - A C nyelvben egyedül egy ilyen van és ?: (kérdőjel kettőspont) operátornak is szokás hívni.

```
C ? T : F
```



Működése a következő, a kiértékeli a legelső (**C**) kifejezést ami igaz vagy hamis értéket adhat és amennyiben ez igaz, kiértékeli a második (**T**) kifejezést ami aztán a kifejezés értékévé (kimenetté) fog válni, ha viszont az első (**C**) kifejezés értéke hamis akkor a harmadik (**F**) kifejezést fogja kiértékelni és ez lesz az operátor kimeneti értéke. Röviden az **if** -hez hasonlóan **T** és **F** közül választ és a kettő közül csak az egyiket értékeli ki.

Pl:

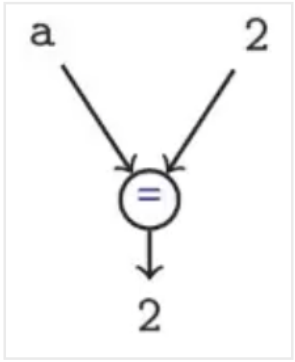
```
/* b  = abs(a) */
b = a < 0 ? -a : a;
```

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

  - Fontos megjegyezni, hogy nem helyettesíti az **if** utasítást, mivel az **if** két utasítás közül, míg a **ternáris** operátor két kifejezés közül választ.

- **Operátorok mellékhatásai:**
  - Azt tudjuk, hogy függvényeknek lehet főhatásuk és mellékhatásuk. Van egy főhatása, ami az, hogy a függvény (operátor) kiszámolja és vissza adja a visszatérési értékét, mellékhatás az lehet hogy a függvény még ezen felül csinál még valamit, pl: globális

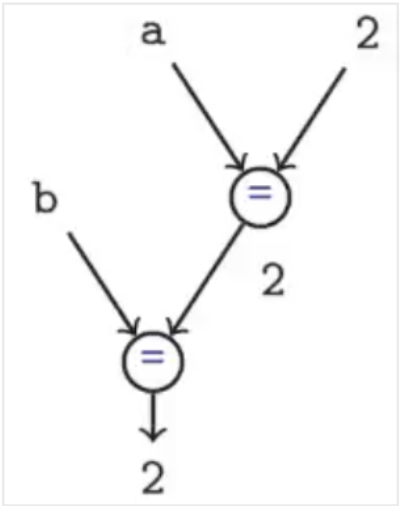
változóhoz nyúl, standard inputról olvas, standard outputra ír, stb.
- Operátorok kiértékelésének is lehet mellékhatása, ilyen az ha egy operátor valamelyik argumentumát, valamelyik operandusát megváltoztatja.
- Erre a tökéletes példa az értékadás operátor (=), ez azt jelenti hogy C -ben az értékadás kifejezés, aminek a mellékhatása az értékadás (a megváltozik), főhatása az új értéke.



- Eddig az értékadást utasításként használtuk a programokban, de lehet az értékét is használni.

Pl:
A főhatás miatt ez is értelmes:

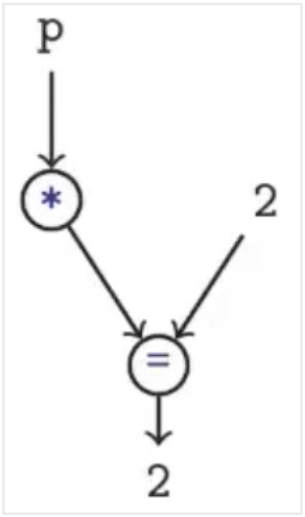
```
b = a = 2
/* b = (a  = b) */
```



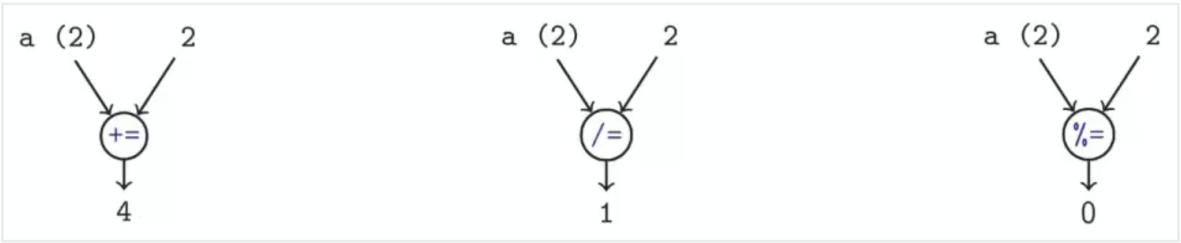
- **Balérték (lvalue):**

- A bal oldali operandusa csak olyan kifejezés lehet ami megváltoztatható.
- Balérték (lvalue) egy olyan kifejezés, amely értékadás bal oldalán állhat, ez lehet:

```
változóhivatkozás          a    = 2
tömbelem                   t[3] = 2
dereferált mutató          *p   = 2
struktúratag               v.k  = 2
struktúratag               q->x = 2
```



- **<u>Viszonyított értékadás:</u>**
  - Ezek mind úgy dolgoznak, hogy a bal oldali operandusnak balértéknek kell lennie, a jobb oldali tetszőleges kifejezés lehet.



  - Ezt körülbelül felfoghatjuk úgy, hogy:

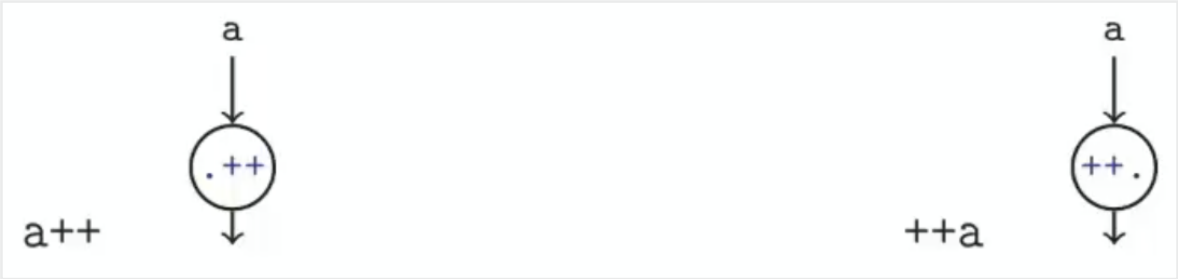
```
<balérték> = <balérték> <operátor> <kifejezés>
```

Pl:

```
a += 2;          /* a = a + 2;
                    */
t[rand()] += 2; /* NEM EGYENLŐ AZZAL, HOGY t[rand()] = t[rand()] + 2; */
```

- A viszonyított értékadás a balértéket csak egyszer értékeli ki.

- **Egyéb mellékhatásos operátorok:**



a++ a -t növeli eggyel, visszaadja a eredeti értékét (posztinkremens operátor).
++a a -t növeli eggyel, visszaadja a új értékét (preinkremens operátor).

Pl:

```
    b = a++;          /* b = a;      a += 1;
posztinkremens */
    b = ++a;          /* a += 1;     b = a;
preinkremens   */
```

```
    /* tömb feltöltése */
    int i = 0, a;
    while (scanf("d%", &a) == 1)
        t[i++] = a;          /* { t[i] = a; i++; }
*/
```

- Mellékhatásos kifejezések utasításként is szerepelhet a programban:
- Kifejezésutasítás:
  `<Kifejezés>;`
- A kifejezést kiértékeljük és az értékét eldobjuk.

Pl:

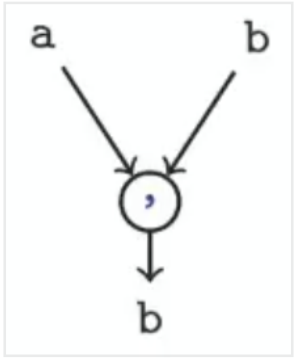
```
    a = 2;          /* kifejezésutasítások
*/
    i++;            /* a főhatást elnyomjuk
*/
    b %= 8;         /* a mellékhatás
utasításrangra emelkedik     */
```

- Mivel a főhatást elnyomjuk, csak mellékhatásos kifejezésekből van értelme kifejezésutasításokat alkotni.

Pl:

```
    2 + 3           /* helyes utasítás, semmit
nem hajt végre */
```

- **A vessző operátor:**



- Az operandusokat balról jobbra értékeli ki, az első (**a**) kifejezés értékét eldobjuk, így a teljes kifejezés értéke és típusa a második (**b**) kifejezés értéke illetve típusa lesz. Ez azt jelenti hogy csak akkor lesz értelme ennek az operátornak, ha a bal oldali operandus kiértékelésekor történik valami, egy mellékhatás lezajlik, tehát a bal oldali operandusnak mellékhatásosnak kell lennie.

Pl:

```
    /* A kétjegyűek növekvő lépésközzel */
    for (step = 1, j = 10; j < 100; j += step;
step++)
      printf("%d\n", j);
```

- **Definiálatlan kiértékelési sorrend:**
  - Az operandusok kiértékelési sorrendje a legtöbb operátor esetén definiálatlan! A definiálatlan működés egy “véletlen program” -hoz vezet.

  Pl:

```
n = i + i++;
a[i] = i++;
f(++i, ++i);
```

  - Ilyenkor a fordító program bármilyen sorrendben elvégezheti a műveleteket, ebbe nekünk nincs beleszólásunk. Erre az egyetlen megoldás, hogy ilyet soha nem írunk le C programban.
  - Mindig igyekezni kell szétválasztani a kifejezésben megjelenő fő- és mellékhatást!

Szöveges fájlok:

- Szöveges fájlok kezeléséhez az **#include <stdio.h>** könyvtárat kell használjuk.
- **Szöveges fájlba való írás és olvasás:**
  - Készítenünk kell egy **FILE típusú pointe**rt amivel majd kezelni tudjuk a fájl tartalmát.
  - A fájlt az **fopen()** függvénnyel lehet megnyitni, ami kettő paramétert kér, az első a **fájl neve**, a második a **megnyitás módja** (w = write, r = read), a függvény egy pointert fog vissza adni, amivel a továbbiakban azonosítani tudjuk majd a fájlt. Amennyiben nem létezik a file amibe írni akarunk az a program lefutásakor magától létre fog jönni.
  - A már megnyitott fájlba az **fprintf()** függvényel lehet írni, a függvénynek meg kell adni, első paraméterként annak a **fájlnak a pointerének a nevét**, amibe írni kívánunk majd pedig azt **amit bele szeretnénk írni**. Amennyiben egymás után több sort szeretnénk beírni a fájlba azt ugyan úgy tehetjük meg mint egy **printf()** függvénynél.

- Fájlból olvasni az **fscanf()** függvénnyel lehet, ami paraméternek a **fájlra mutató pointert**, a **beolvasandó értékek karakter kódját**, azt, hogy **mindezt hol szeretnénk eltárolni** (ha több dologról van szó akkor vesszővel elválasztva kell írni őket). Ugyan az a visszatérési értéke mint a **scanf()** függvénynek, tehát akkora számmal fog visszatéri ahány a minta szerinti karaktert sikeresen beolvasott, ha kevesebbet talál az érték kevesebb is lehet, ha azonnal a **fájlvég**ével találkozik akkor **-1** lesz az értéke.
- A fájl bezárásához az **fclose()** függvényt kell használni, ami a megnyitott **file pointer**ének nevét kéri paraméterként.

Pl:

Írás:

```
#include <stdio.h>

int main(void) {
     FILE *f;
     f = fopen("fajl.txt", "w");
     fprintf(f, "Hello\n");
     fclose(f);
     return 0;
}
```

Olvasás:

```
#include <stdio.h>

int main(void) {
     int a;
     char str[100];
     double d;
     FILE *f;
     f = fopen("fajl.txt", "r");
     fscanf(f, "%s%d%lf", str, &a, &d);
```

```
    fclose(f);
    return 0;
}
```

- Az **fprintf()** és **printf()** függvényeknek van egy visszatérési értéke ami egy egész érték és azt adja meg hogy hány karakter íródott a fájlba a függvény meghívásakor.
- Ha a programon belül írásra megnyitunk egy fájlt akkor a program teljesen törli a tartalmát akár volt akár nem.
- Fájlba írás és olvasás nem azonnal történik hanem egy ún. **buffer** tárolón keresztül történik, ilyenkor bármi amit írnunk vagy olvasunk az egy a rendszer által lefoglalt ideiglenes tároló helyre, egy ideiglenes tömbbe kerül, innen akkor fog bekerülni a fájlba vagy a tömbbe/változóba amibe olvasunk, ha az általunk írni vagy olvasni kívánt karakterek mérete meghaladja a buffer méretét, amikor az írás vagy olvasás megtörténik, a buffer törlődik és ugyan ez kezdődik előröl vagy ha hamarabb lezárjuk a fájlt a **fclose()** függvénnyel ugyan ez fog történni. Ezért az **fclose()** függvény elengedhetetlenül fontos, mert az írni vagy olvasni kívánt dolgok ilyekor kerülnek ténylegesen a fájlba vagy a tömbbe/változóba.
- Az **fclose()** függvénynek van egy státusz jelzése arra, hogy sikerült-e a fájlba írás vagy sem. Ha **sikerült akkor ez 0**, **ha nem** sikerült **akkor valami más**. Ezt mindig ellenőrizni kell:

  Pl:

  ```
  if (fclose(f) != 0)
     return 1;
  ```

- A fájl megnyitásánál ugyan ilyen fontos, hogy megnézzük van-e írási és vagy olvasási jogom, emiatt az **fopen()** függvény visszatérési értékét is nézni kell, **ha nem sikerül a fájlt írásra és vagy olvasásra megnyitni akkor NULL pointer**t ad majd vissza.

  Pl:

  ```
  if (f == NULL)
  ```

```
    return 1;
```

Dinamikus adatszerkezetek I - egyszerű listák (listák és vermek):

- A verem egy olyan adatszerkezet amiben adatokat tudunk tárolni, ez egy LIFO (Last In First Out) jellegű tároló, tehát ha bele teszek egy elemet akit utoljára tettem bele ahhoz fogok tudni először hozzáférni. A vermet mi láncolt listával valósítjuk meg.

Pl:

```
/* használhatunk saját vagy már létező típusú
adat fajtát is */
typedef int data_t;

/* csinálunk egy típusnévvel ellátott struktúrát
*/
typedef struct list_elem {
    /* a data nevű változóban lesz a tényleges
adat */
    data_t data;
    /* ez a pointer pedig a következő list_elem
típusú struktúrára mutat */
    struct list_elem * next;
/* a típusnév list_elme lesz és azonnal
csinálunk belőle egy list_ptr nevű pointert is
ami erre a struktúrára fog mutatni */
} list_elem, *list_ptr;
```

- Itt használni fogunk egy **->** mutatót, aminek a működése a következő, ez a nyíl ugyan azt a szerepet látja el mint egyes programnyelvekben a pont, amikor egy tömbön belüli dologhoz akarunk hozzáférni, ha van egy pointerünk amin belül van egy belső valami akkor, azt a valamit fogja vissza adni.

Például az alul látható adatszerkezetből szeretnének megtudni a data nevű változó értékét

```
list_ptr head; /* csinálunk egy list_ptr típusú
```

```
pointert */
printf("%d", head->data); /* elkérjük a head
pointer által mutatott adatszerkezet data
elemének az értékét */
```

- Ennek az adatszerkezetnek 3 művelete van:
  - **"push"** művelettel a verem tetejére tudunk elhelyezni egy új elemet.

```
    /* vissza fogja adni a verem új tetjének a
címét és meg kell neki adni a verem tetejének
jelenlegi címét illetve azt az adatot amit bele
akarunk illeszteni a verembe */
    list_ptr push(list_ptr head, data_t d) {
        /* csinálunk egy list_ptr típusú
pointert aminek memóriát foglalunk aminek
list_elem méretűnek kell lennie */
        list_ptr p = malloc(sizeof(list_elem));
        /* a foglalt területre bemásolom a
kapott d adatot */
        p->data = d;
        /* most az új elemet hozzá kell
láncoljuk a már meglévőkhöz */
        p->next = head;
        /* végül meg kell változtatni a verem
kezdőcímét a p pointer értékére */
        head = p;
        /* ezt egy return-el vissza is lehet
adni */
        return p;
    }
```

  - **"pop"** művelettel az épp a verem tetején lévő elemet tudjuk levenni.

```
    /* vissza fogja adni a verem új kezdőcímét,
paraméterként pedig az eddigi címet kéri */
    list_ptr pop(list_ptr head) {
        /* először meg kell nézni, hogy nem e
üres a lista amiből poppolni akarunk, amenniben
az egyszerűen egy NULL pointert adunk vissza */
        if (head == NULL)
```

```
            return NULL;
        /* csinálunk egy ideiglenes pointert ami
eltárolja a verem következő elemének a címét
hogy az ne vesszen el */
        list p = head->next;
        /* ez után felszabadíthatjuk a verem
tetején lévő memóriát */
        free(head);
        /* azért hogy a head pointer ami az
előző legfelső elem címe volt és már nem
létezik, ezért felül kell írni a p vel ami a
verem következő elemének címét tárolja */
        head = p;
        /* az így keletkezett lista fejet vissza
adjuk */
        return head;
    }
```

- **“top”** művelettel a verem legtetején lévő elem értékét kapjuk meg.

```
    /* vissza adja a veremben tárolt adattípust
és paraméterként kap egy a verem tetejére mutató
lista pointert*/
    data_t top(list_ptr head) {
        /* a head pointer által mutatott adatot
(data) adja vissza */
        return head->data:
    }
```

Példa a fő programra:

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


int main(void) {
    /* csinálunk egy láncolt lista pointert,
amit stacknek nevezünk el és alapból NULL
pointernek állítjuk be ezzel jelezve hogy a
```

```
láncolt listának nincs egy eleme se */
    list_prt stack = NULL;

    /* a push művelettel bele rakunk egy a stack
pointer által rámutatott listába egy elemet
push(pointer neve, adat); */
    /* Ha bele rakunk egy új elemet a verembe a
8 -ast pl

    8-> 4-> 3-> 5
        ^
    akkor ő a verem eddigi kezdetére fog mutatni
(most a 4), de most még nem ő a verem feje,
hanem a 4 -es, ahhoz hogy ez megoldjuk a verem
tetejére mutató pointert át kell állítsuk az új
elemre, ezért ha push művelettel betolunk egy új
elemet a verem elejébe akkor a verem
kezdőcímének meg kell változnia.

    stack = push(stack, 8);
    8-> 4-> 3-> 5
    ^
    */

    stack = push(stack, 5);
    stack = push(stack, 3);
    stack = push(stack, 4);

    /* miután bele raktunk valakit megnézhetjük
a top függvénnyel hogy épp ki van a verem
tetején top(pointer név); */

    printf("%", top(stack));

    /* miután raktunk elemeket a listába azokat
a pop függvénnyel tudjuk eltávolítani belőle
pop(stack);, viszont a pushhoz hasonlóan ez is
meg fogja változtatni a verem kezdőcímét, ezért
itt is át kell venni a pop-tól a verem tetejének
új kezdőcímét */

    stack = pop(stack);
```

```
    stack = pop(stack);
    stack = pop(stack);

    return 0;
}
```

Dinamikus adatszerkezetek II - strázsás és fésűs listák:

Bináris fájlok:

- A bináris fájlok kezelése hasonlóan történik a sima szöveges fájlokéhoz.
- Egy fájlt az **fopen(fájlnév, "wb");** funkcióval tudunk bináris írás módban (**wb**) megnyitni ezután ezt még oda kell adni egy **FILE** típusú pointernek.
- Egy fájlba a következő módon lehet írni, **fwrite(kiírandó tömb címe, a tömb egy elemének mérete bájtokban, a tömb elemeinek száma, a fájl amibe írni akarunk);**, ha tömb helyett csak egy sima változót akarunk kiiratni akkor a változó neve elé kell rakni az **&** jelet (ami a címképzés operátor), mivel az **fwrite();** függvény egy címet kér első paraméterként.
- Ha csak egy darab karaktert akarunk kiírni egy fájlba akkor érdemes az **fputc('karakter', fájl pointer);** funkciót használni.
- Mi most itt little at the end az az kis végződésű bájt sorrendel dolgozunk és általában ez a standard, de van rá esély hogy más sorrendet használ egy fájl így mindig biztosra kell menni mielőtt külső fájlal kezdünk dolgozni.
- Ahhoz hogy a bináris fájlból ki tudjunk olvasni adatokat ismét az fopen funkciót kell használjuk de most bináris írás mód helyett (**wb**), bináris olvasás (**rb**) módban kell megnyitni vele a fájlt, ez után az **fread(amibe olvasunk, adat fajta mérete bájtba, elemek száma, melyik fájlból);** funkcióval tudunk kiolvasni a fájlból.

- Amikor bináris fájlba írunk, majd olvasunk ki belőle, pontosan ugyan azt fogjuk megkapni.
- Mind az **fwrite()** mind pedig az fread() funkciónak van visszatérési értéke ami egy egész szám. Ez az **fwrite()** esetében azt mondja meg hogy a kiírni kívánt elemekből hányat sikerült ténylegesen kiírni. Az **fread()** esetében pedig azt adja vissza hogy a beolvasni kívánt elemek közül hány darabot sikerült tényelegesen beolvasni.

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


int main(void) {
    int t[4] = {1, 2, 256, 4096};
    int t2[4];
    double d = 8.2;
    double d2;
    char s[10];
    char s2[10];
    srtcpy(s, “alma”);
    int db;

    FILE *f;
    f = fopen("file.bin", "wb");
    db = fwrite(t, sizeof(int), 4, f);
    db = fwrite(&d, sizeof(double), 1, f);
    db = fwrite(s, sizeof(char), 10, f);
    fclose(f);

    f = fopen("file.bin", "rb");
    db = fread(t2, sizeof(int), 4, f);
    db = fread(d2, sizeof(double), 1, f);
    db = fread(s2, sizeof(char), 10, f);
    fclose(f);

    return 0;
}
```

Bináris vs szöveges fájlok:

- Amennyiben hordozható programokat akarunk írni, mindig át kell gondoljuk hogy egy fájlt szöveges vagy bináris formában fogunk kinyitni, mivel pl. a Windows -nál egy sortörés (**\n**) jel kettő bájtot foglal el egy szöveges fájlban, ez más operációs rendszereken mindössze egy bájt és ez komoly gondokat tud okozni, ha nem figyelünk rá.

Többdimenziós tömbök:

```
int (*t)[2];
```

- A **t** egy **2** elemű **int** tömbre mutató pointer.
- A tömbre mutató pointer és a tömb első elemére egy **int** -re mutató pointer között az a különbség, hogy ha például egy tömbre mutató pointert balról megcsillagozok akkor az eredménynek lehet képezni az elemeit, tehát indexelhető lesz, ha egy **int** -re mutató pointert megcsillagozok akkor egy **int** -et kapok mivel az nem indexelhető. A pointer aritmetika miatt ha ezt a **t** -t meg növelem egyel, akkor a benne tárolt bájtokban mért érték az két **int** -nyivel fog nőni, mert neki már a következő két elemű tömbre kell mutatnia a memóriában. Ebből az is következik hogy ha van egy másik, tömbre mutató pointerünk aminek az előzőtől eltér az elemszáma, akkor a kettő tömb típusa különbözni fog. Szimplán csak tömbre mutató pointer nem létezhet, csak fordítási időben meghatározott méretű tömbre mutató pointer létezhet.
- Ha van egy **int a[3][2];** kifejezésünk azt úgy kell értelmezni, hogy **a** -nak az elemének az elemének a típusa **int**. Ilyenkor voltaképpen egy tömbön belül hozunk létre tömböket.

```
int a[3][2] = {{1, 2}, {11, 12}, {21, 22}};
```

- Csak olyan többdimenziós tömböt tudunk létrehozni aminek az elemei azonos típusúak és aminek az összes belső tömbje azonos méretű.

- Az ilyen tömböket az alábbi módon lehet bejárni, adott esetben 0 -kkal feltölteni:

```
for (int i = 0; i < 3; i++)
  for (j = 0; j < 2; j++)
     a[i][j] = 0;
```

- Ha ebből a nullázó bejárásból függvényt akarunk csinálni, akkor ahhoz a függvénynek paraméterként normális esetben a tömb kezdőcímét kell oda adni és a tömb méretét, ezzel szemben mivel itt az a tömbben 2 elemű int tömbök vannak ezért ezt kell oda adjuk a függvénynek, illetve továbbra is oda kell adjuk a tömb elemeinek a számát. Nagyon fontos hogy a függvénynek adott pointer a típusában kell tárolja a belső méretet, ez azért fontos mert amikor a for ciklusban az i indexet tudjuk alkalmazni, ahhoz tudni kell hogy a pointer megnövelésekor pontosan hány bájttal kell annak ugrania.

```
void nullazo(int t[][2], int n) {
    for (int i = 0; i < n; i++)
            for (j = 0; j < 2; j++)
               a[i][j] = 0;
}
```

- Ha ugyan erre a célra egy általánosítottabb függvényt akarunk írni amivel szabadon megadható méretű több dimenziós tömböket lehet bejárni az az alábbi módon egy kis trükkel megvalósítható. A trükk az hogy ha a memóriában elhelyezek egymás mellett x darab y elemű valamilyen típusú tömböt, akkor igazából a memóriában 6 darab olyan típusú változót helyeztem el egymás mellett. Emiatt ha képezzük ennek a többdimenziós tömb nulladik résztömbjének a nulladik elemének a címét, majd ezt a címet átadjuk egy pointernek akkor az a pointer olyan lesz mint ha egy sima tömbnek a nulladik elemének a címét tárolná.

```
int *p = &a[0][0];
```

- Ilyenkor már ezt a címet használhatjuk úgy mint ha egy sima tömb lenne és ezt átadhatjuk egy függvénynek az x és y elemszámok mellett. Viszont itt már nem használhatunk

kétszeres indexelést (**t[i][j]**), ehelyett most a fordító helyett a függvény fogja kiszámolni az x sor y oszlopának elemét (**i * m + j**)

```
void nullazo2(int t[], int n, int m) {
      for (int i = 0; i < n; i++)
            for (j = 0; j < m; j++)
               a[i * m + j] = 0;
 }
```

- Ebben az esetben ahelyett, hogy a **nullazo2();** függvénynek a **(p, x, y)** paramétereket adnánk, megtehetjük azt is, hogy a **p** pointer helyett egy kényszerített típus konverziót használunk amivel adott esetben az **a** kétdimenziós tömböt **(int *)** pointerré tesszük és ezt adjuk oda a függvénynek **nullazo2((int *)a, x, y);**
- Az utolsó ide tartozó dolog a dinamikus többdimenziós tömb foglalása, ehhez először is a hagyományos módon foglalnunk kell egy akkora egydimenziós tömböt, amibe a tömb összes eleme bele fog férni és ez után ezt az előbb bemutatott módon az indexelés átértelmezésével fogom tudni kezelni. Itt most az x és y a többdimenziós tömb méreteit adja meg.

  Pl:

```
int x = 5, y = 8;
int *p = malloc(x * y * sizeof(int));

nullazo2(p, x, y);

free(p);
```

Rekurzió:

- A rekurzió egyszerűsített fogalma, hogy egy definíció vissza hivatkozik önmagára.
- Általában azt mondhatjuk hogy ha egy problémát rekurzívan fogalmazunk meg akkor azt vissza vezethetjük egy hasonló egyszerűbb problémá(k)ra, emellett meg kell adjuk a triviális eseteket is, ezek lesznek a rekurzió megállási feltételei, melyek voltaképpen megakadályozzák azt, hogy egy

végtelenített visszavezetési láncot kapjunk.
- A rekurzióval nagyon elegáns, de bizonyos estekben nagyon pazarló lehet.
  - Például, ha egy rekurzív függvénnyel akarjuk kiszámolni egy szám faktoriálisát, akkor mivel minden n-1 -edik esetben újra meg lesz hívva a függvény, ami miatt feleslegesen részeredményeket tárolunk el, illetve feleslegesen hívunk meg egy csomó függvényt.

Faktoriális

$$n! = \begin{cases} (n-1)! \cdot n & n > 0 \\ 1 & n = 0 \end{cases}$$

```
unsigned fact_rec(unsigned n) {
   if(n == 0)
      return 1;
   return fact_rec(n-1) * n;
}
```

  - Ha ehelyett a faktoriálist iterációval számoljuk ki, az kevésbé lesz elegáns, viszont sokkal hatékonyabb lesz mint a rekurzív megvalósítás.

```
unsigned fact_iter(unsigned n) {
   unsigned f = 1, i;
   for (i = 2; i <= n; ++i)
      f *= i;
   return f;
}
```

Rekurzió vagy iteráció:
- Ha arról kell döntsünk, hogy rekurzióval vagy iterációval akarunk megoldani egy feladatot, az alábbi dolgokat kell észben tartsuk:
  - Minden rekurzív algoritmus megoldható iterációval (ciklusokkal) viszont a két megoldás nem lesz pontosan ugyan az, ezért nehéz lehet egyikből másikba átírni egy ilyen algoritmust.
  - Minden iterációval megoldható algoritmus megoldható rekurzívan, ilyenkor általában könnyen tudunk az egyik megoldás fajtából a másikra változtatni, de ez sokszor nem hatékony.

    - Mindig a problémától függ, hogy melyik módszert érdemes használni.
- Ha például egy függvényt akarunk írni, ami kap egy számot (**n**) és egy számrendszert (**base**) és kiírja hogy az adott **n** szám hogy néz ki a **base** számrendszerben, akkor már érdemes elgondolkodni azon, hogy rekurziót használjunk. A példában jobban megéri rekurziót használni mint iterálást.

  Rekurzívan:

```
  void print_base_rec(unsigned n, unsigned
base) {
    if (n >= base)
       print_base_rec(n/base, base);
    printf("%d", n%base);
  }
```

  Iterációval:

```
  void print_base_iter(unsigned n, unsigned
base) {
      unsigned d; /* n-nél nem nagyobb base-
hatvány */
      for(d = 1; d*base <= n; d *= base);
      while(d > 0) {
          printf("%d", (d/n)%base);
          d /= base;
      }
  }
```

Közvetett rekurzió:

- A rekurzió lehet közvetlen és közvetett is, ha közvetett akkor nem magát hívja meg hanem egy másik függvényt, ami pedig majd meghívja őt, ha pedig közvetlen rekurzióról beszélünk akkor egy függvény önmagát hívja meg.
- Abban az esetben, ha közvetett rekurziót akarunk használni, ahhoz hogy elkerüljünk egy paradoxont először deklarálnunk kell, azt a függvényt amit az először meghívott függvény meg fog hívni, ezt egyszerűen úgy tudjuk megcsinálni hogy egy sorba leírjuk a függvény visszatérési értékét, nevét, és a paramétereinek típusait.

Pl:

```
void b(int);

void a(int n) {
  …
  b(n);
  …
}

void b(int n) {
  …
  a(n);
  …
}
```

- Elődeklaráció közvetve rekurzív adatszerkezetek esetén is szükséges

Pl:

```
struct child_s; /* deklaráció */

struct mother_s {
  char name[50];
  struct child_s *children[20]; /* a
gyerekek pointer tömbje */
}

struct child_s {
  char name[50];
  struct mother_s *mother; /* mutató az
anyára */
}
```

Gyorsrendezés (quick sort) algoritmus:

- Helyben szétválogatáson alapul, ami nem más, mint hogy n elemű tömböt helyben szétválogatunk úgy, hogy a vezérelemnél (“pivot”) kisebb elemek elemek a tömb elejére kerülnek. Ez egy az oszd meg és uralkodj és rekurzív

hívásokra alapuló gyorsrendező algoritmus.

Pszeudokód:

```
i <- 0; j <- n;
AMÍG i < j
   HA t[i] < pivot
      i <- i+1;
   EGYÉBKÉNT
      j <- j-1;
      HA t[j] <= pivot
         t[i] <-> t[j]
```

- Ezzel az algoritmussal az n elemű tömböt szét tudtuk válogatni “kis elemek” - pivot - “nagy elemek” tömbökre
- Válogassuk szét külön-külön az i elemű “kis elemek” és az n-i-1 elemű “nagy elemek” tömböt ugyanezzel a módszerrel. Ez a rekurzív ciklus akkor fog leállni a tömb egyelemű lesz, mivel az már rendezett.
- Az algoritmus lépésszáma, az első körben n lépés, a második körben i + (n - i - 1) = n - 1 lépés és ezzel a körök száma kb log(2)n (átlagosan minden tömböt sikerül felezni), így a rendezés n*log(2)n lépésben megoldható.

```
void qsort(int array[], int n) {
  int pivot = array[n/2];
  int i = 0, j = n;
  while(i < j) {
     if(array[i] < pivot)
        i++;
     else {
        j--;
        if(array[j] <= pivot)
           swap(array+i, array+j);
     }
  }

  if(i > 1)
     qsort(array, i);
  if(n-i-1 > 1)
     qsort(array+i+1, n-i-1);
}
```

- Itt az utolsó kettő if -el azt ellenőrizzük hogy a bal illetve jobb oldalnak megfelelő tömbök nagyobbak e mint egy és ha igen meg hívjuk az adott oldalra a függvényt.

Fák:

-