Programozás 1

Adattípusok

ADATTÍPUSOK

Adattípusok

- Az adatkezelés szintjei egyúttal különböző absztrakciós szinteknek felelnek meg.
  - Probléma szintje
  - Szaktudományos és matematikai szint
  - Absztrakt adattípus szint
  - Virtuális adattípus szint
  - Fizikai adattípus szint

Adattípusok

- Probléma szintje.
  - A felhasználó ezen a szinten fogalmazza meg a megoldandó problémát, a bemenő és kimenő adatokat, valamint a bemeneti-kimeneti feltételt.

Adattípusok

- Szaktudományos és matematikai szint.
  - A probléma megoldásához a kiindulópontot a szaktudomány és/vagy matematikai modell képezi. Ezen a szinten azonban arra a kérdésre kapunk választ, hogy milyen adatok szerepelnek a problémában és hogy milyen összefüggések érvényesülnek. A hogyan kérdésre az algoritmustervezés során kell válaszolni.

Adattípusok

- Absztrakt adattípus szint.
  - Az algoritmustervezés során határozzuk meg, hogy milyen műveleteket kell alkalmazni a problémában szereplő adatokon ahhoz, hogy a megoldást megkapjuk. Ezt a szintet azért nevezzük absztrakt szintnek, mert még nem döntöttünk arról, hogy az adatokat hogyan tároljuk és a műveleteket milyen algoritmusok valósítják meg. A tárolásról és a műveletek megvalósításáról csak akkor szabad dönteni, amikor a tervezésben az összes információ rendelkezésünkre áll ahhoz, hogy a legmegfelelőbb, leghatékonyabb megvalósítást kiválasszuk.

Adattípusok

- Virtuális adattípus szint
  - A probléma megoldását adó algoritmust valamely programozási nyelven írjuk le. A programozási nyelvet virtuális gépnek tekinthetjük, amely rendelkezik eleve adott adattípusokkal és típusképzésekkel. Az absztrakt adattípusokat tehát virtuális adattípusok felhasználásával valósítjuk meg.

Adattípusok

- Fizikai adattípus szint
  - Az adatokat végső soron fizikai eszköz, a memória tárolja és a műveleteket fizikai gépi műveletek valósítják meg. A memória a legtöbb gép esetében byte-ok véges sorozata, tehát végső soron minden adattípust e lineáris szerkezetű tárolóval kell reprezentálni.

Adattípusok

- A C nyelv virtuális adattípusait fogjuk áttekinteni, azonban az adattípusokat a legtöbb esetben mint absztrakt adattípusokat vezetjük be.
- Ennek az az oka, hogy ezek olyan általános programozási fogalmak, amelyek akkor is használhatók algoritmusok tervezése során, ha a megvalósítás nyelve nem a C.
- Ekkor a választott nyelv virtuális (vagy gépi) adattípusainak felhasználásával kell az adattípusokat megvalósítani.

Adattípusok

- A C (és sok más hasonló programozási) nyelv alapvető tulajdonsága, hogy rendelkezik néhány eleve definiált adattípussal továbbá típusképzési mechanizmusokkal.
- Így a már (eleve vagy a programozó által) definiált adattípusok felhasználásával újabbakat definiálhatunk.

Elemi adattípusok

- Az adattípusokat osztályozhatjuk aszerint is, hogy az értékhalmazuk elemei összetettek-e vagy elemiek.
- Elemi adattípusok
  - Egész típusok (int, signed, unsigned, short, long)
  - Karakter típus (char, signed, unsigned)
  - Felsorolás (enum típusképzéssel)
  - Valós típusok (float, double, long)
  - Logikai típus (C-ben nincs közvetlen megvalósítása)

A C nyelv elemi adattípusai

- Az elemi és módosított típusok.

C típus	méret(bájt)	alsó határ	felső határ
char	1	?	?
signed char	1	-128	127
unsigned char	1	0	255
short int	2	-32768	32767
unsigned short int	2	0	65535
int	4	-2147483648	2147483647
unsigned int	4	0	4294967295
long int	4	-2147483648	2147483647
unsigned long int	4	0	4294967295
long long	8	-2^{63}	$2^{63}-1$
float	4	-+3.4028234663852886E+38	
double	8	-+1.7976931348623157E+308	
long double	8	-+1.7976931348623157E+308	

Egész típusok a C nyelvben

- Egy n bites tárterületnek 2^n állapota van. Az értékhalmaz szokásos kijelölése a következő:
  - Ha negatív számok nem szerepelnek az értékhalmazban, akkor az értékhalmaz $0..2^n-1$
  - Ha szerepelnek, akkor pedig $-2^{n-1}..2^{n-1}-1$

- A C nyelv különféle egész adattípusai az értékhalmazukban különböznek egymástól, az értelmezett műveletükben megegyeznek.

Egész típusok a C nyelvben

- Egész kifejezésben bármely egész típusú tényező (akár vegyesen többféle is) szerepelhet.
- Egész konstans típusa az az egész típus, amely a legszűkebb olyan értékhalmazú, amelynek eleme a kifejezés értéke.
- Értékadó művelet jobb oldalán álló kifejezés kiértékelése független attól, hogy a bal oldalon milyen típusú változó van.

A C nyelv elemi adattípusai

- A módosítók
  - signed
    - előjeles
  - unsigned
    - Előjel nélküli
  - short
    - rövid
  - long
    - hosszú

A C nyelv elemi adattípusai

- Az egyes gépeken az egyes típusok mérete más-más lehet, de minden C megvalósításban

```
sizeof(short)<=sizeof(int)
                sizeof(int)<=sizeof(long)
sizeof(float)<=sizeof(double)
             sizeof(double)<=sizeof(long double)
```

- Az **`int`** elhagyható (de nem fogjuk megtenni) ugyanis mindenütt, ahol típust kell megadni, de ezt elfelejtettük, oda **`int`**-et gondol a fordító.

A C nyelv elemi adattípusai

- A limits.h-ban sok konstans van deklarálva, ha ezeket használni szeretnénk, akkor

```
#include <limits.h>
```

sort kell beszúrni valahová a program elejére.

A C nyelv elemi adattípusai

- Néhány konstans a limits.h-ból

```
#define CHAR_BIT                  8   /* "char" */
#define SCHAR_MIN              (-128)   /* "signed char" */
#define SCHAR_MAX                127   /* "signed char" */
#define UCHAR_MAX                255   /* "unsigned char" */
#define CHAR_MIN           SCHAR_MIN   /* "char" */
#define CHAR_MAX           SCHAR_MAX   /* "char" */
#define SHRT_MIN            (-32768)   /* "short int" */
#define SHRT_MAX              32767   /* "short int" */
#define USHRT_MAX             65535   /* "unsigned short int" */
#define INT_MIN    (-2147483647-1)   /* "int" */
#define INT_MAX         2147483647   /* "int" */
#define UINT_MAX        4294967295   /* "unsigned int" */
#define LONG_MIN   (-2147483647-1)   /* "long int" */
#define LONG_MAX        2147483647   /* "long int" */
#define ULONG_MAX       4294967295   /* "unsigned long int" */
```

Karakter adattípus a C nyelvben

- A **`char`** adattípus a C nyelv eleve definiált elemi adattípusa, értékkészlete 256 elemet tartalmaz.
- A **`char`** adattípus egészként is használható, de kimeneti eszközön karakterként jelenik meg.
  - Hogy melyik értékhez melyik karakter tartozik, az az alkalmazott kódtáblázattól függ.
  - Bizonyos karakterek (általában a rendezés szerint első néhány) vezérlő karakternek számítanak, és nem megjeleníthetők.

Karakter adattípus a C nyelvben

- A karaktereket aposztrófok (') között kell megadni.
- A speciális karaktereket, illetve magát az aposztrófot (és végső soron tetszőleges karaktert is) escape-szekvenciákkal lehet megadni.
- Az escape-szekvenciákat a \ (backslash) karakterrel kell kezdeni.

Karakter adattípus a C nyelvben

- Karakterek escape-szekvenciákkal.

újsor	**NL (LF)**	**'\n'**
vízszintes tab	**HT**	**'\t'**
vissza-szóköz	**BS**	**'\b'**
kocsi-vissza	**CR**	**'\r'**
lapdobás	**FF**	**'\f'**
fordított törtvonal	****	**'\\'**
aposztróf	**'**	**'\''**
tetszőleges karakter		**'\ddd'**

Karakter adattípus a C nyelvben

- A **`\ddd`** escape-szekvencia egy fordított törtvonalat és 1, 2 vagy 3 rákövetkező oktális számjegyet tartalmaz, amelyek a kívánt karakter értékét határozzák meg.
  - E konstrukció speciális esete a **`\0`**, amely a NUL karaktert jelöli.
- Ha a fordított törtvonalat követő karakter nem az előbbiek egyike, a fordító a fordított törtvonalat nem veszi figyelembe.

Karakter adattípus a C nyelven

- A kódtábláról csak annyit tételezhetünk fel, hogy
  - 'a'<'b'< ... <'z', az angol ábécé 26 kisbetűjére
  - 'A'<'B'< ... <'Z', az angol ábécé 26 nagybetűjére
  - A számjegy karakterek ténylegesen rákövetkezők, azaz '0'+1=='1', …, '8'+1=='9'
- Mint említettük a **`char`** adattípus egészként is használható. A konverzió a kétfajta típusú érték között automatikus ezért például:
  - **`'\nnn' == 0nnn`**

Karakter adattípus a C nyelven

- Konvertáljunk egy tetszőleges számjegy karaktert (ch) a neki megfelelő egész számmá és egy egyjegyű egészet (i) karakterré.

```
ch - '0'
i + '0'
```

- Konvertáljunk kisbetűt nagybetűvé és nagybetűt kisbetűvé.

```
ch - 'a' + 'A'
ch - 'A' + 'a'
```

Bitenkénti logikai műveletek

- A C nyelvben több bitmanipulációs operátor van, ezek a **`float`** és **`double`** típusú változókra nem, de **`int`** és **`char`** típusokra alkalmazhatók.
  - **`&`** bitenkénti ÉS,
  - **`|`** bitenkénti megengedő (inkluzív) VAGY,
  - **`^`** bitenkénti kizáró (exkluzív) VAGY,
  - **`<<`** bitléptetés (shift) balra,
  - **`>>`** bitléptetés (shift) jobbra,
  - **`~`** egyes komplemens (egyoperandusú).

Bitenkénti logikai műveletek

- A bitenkénti ÉS operátort gyakran használjuk valamely bithalmaz maszkolására. Például azt, hogy páratlan-e x az

```
((x & 1) == 1)
```

  valósítja meg.
- A bitenkénti VAGY operátorral lehet biteket 1-re állítani:

```
x = x | MASK;
```

  ugyanazokat a biteket állítja 1-be x-ben, mint amelyek 1-be vannak állítva MASK-ban.

Bitenkénti logikai műveletek

- Gondosan meg kell különböztetnünk az **&** és **|** bitenkénti operátorokat az **&&** és **||** logikai műveletektől, amelyek egy igazságérték balról jobbra történő kiértékelését írják elő.
- Ha például x értéke 1 és y értéke 2, akkor a C laza típusossága miatt
  - **`x & y == 0`**
  - **`x && y != 0`**

Bitenkénti logikai műveletek

- A **<<** és **>>** léptető (shift) operátorok bal oldali operandusukon annyi bitléptetést hajtanak végre, ahány bitpozíciót a jobb oldali operandusuk előír:

```
x = x << 2;
```

az x bitjeit két pozícióval balra lépteti, a megürült biteket pedig 0-val tölti fel: ez 4-gyel való szorzással egyenértékű.

Bitenkénti logikai műveletek

- Az **`unsigned`** (előjeltelen) mennyiség jobbra léptetése esetén a felszabaduló bitekre nullák kerülnek.
- Előjeles mennyiség jobbra léptetése esetén bizonyos gépeken a felszabaduló bitekre az előjel kerül (aritmetikai léptetés), más gépeken 0 bitek (logikai léptetés).

Bitenkénti logikai műveletek

- A ~ bináris operátor egész típusú mennyiség 1-es komplemensét képezi, vagyis minden bitet negál.
- Ezt az operátort leggyakrabban olyan kifejezésekben használjuk, mint

```
x & ~077
```

amely x utolsó 6 bitjét 0-ra maszkolja. Vegyük észre, hogy x & ~077 független az x méretétől, és így előnyösebb, mint pl. x & 0177700, amely 16 bites x-et feltételez. A gépfüggetlen alak nem növeli a futási időt, mivel ~077 állandó kifejezés, és így fordítási időben értékelődik ki.

Bitenkénti logikai műveletek

- Példák:
  - **~0x08:**
    - **~((char) 0x08) == 0xf7**
    - **~((short)0x08) == 0xfff7**
    - **~((long) 0x08) == 0xfffffff7**
  - **0x16 ^ 0x3a == 0x2c**
    - **00010110 ^**
      00111010 ==
      00101100

Bitenkénti logikai műveletek

- Példák:
  - **`0x16 & 0x3a == 0x12`**
    - `00010110 &`
      `00111010 ==`
      `00010010`
  - **`0x16 | 0x3a == 0x3e`**
    - `00010110 |`
      `00111010 ==`
      `00111110`

Bitenkénti logikai műveletek

- Példák:
  - `0x96 << 2 == 0x58`
    - `10010110 << 2 == 01011000`
  - `0x16 >> 2 == 0x05`
    - `00010110 >> 2 == 00000101`
  - `0x96 >> 2 == 0x25 / 0xe5`
    - `10010110 >> 2 == 00100101`
    - `10010110 >> 2 == 11100101`

Bitenkénti logikai műveletek

- Van-e előjeles léptetés?

```
char a;
unsigned char b;
a = b = 128;
a >>= 1; b >>= 1;
printf("Ezen a gépen%ssigned char van\n",
        ((a == b) ? " un" : " " ));
```

Bitenkénti logikai műveletek

- Következő példában néhány bitoperátor működését szemléltetjük.
- A `getbits(x, p, n)` függvény x-nek a p-edik pozíción kezdődő n-bites mezőjét adja vissza (jobbra igazítva).
- Feltételezzük, hogy
  - a 0. bitpozíció a jobb szélen van
  - n és p értelmes pozitív értékek.
- Például getbits (x,4,3) a 4, 3 és 2 pozíción levő három bitet szolgáltatja, jobbra igazítva.

Bitenkénti logikai műveletek

```
int getbits(unsigned x, p, n)
{
  return ((x >> (p + 1 - n)) & ~(~0 << n));
}
```

- `x >> (p + 1 - n)`
  - A kívánt mezőt a szó jobb szélére mozgatja.
  - Az x argumentumot unsigned mennyiségnek deklarálva biztosítjuk, hogy a jobbra léptetéskor a felszabaduló bitek biztosan nullákkal töltődjenek fel.

Bitenkénti logikai műveletek

- ~0
  - Csupa 1 bitet jelent
- ~0 << n
  - Utasítás segítségével n bitpozícióval balra léptetve a ~0 értéket a jobb oldali n biten csupa nullákból álló, a többi pozíción egyesekből álló maszk jön létre.
- ~(~0 << n)
  - Olyan maszk keletkezik, amelyben a jobb oldali biteken állnak egyesek.

Bitenkénti logikai műveletek

- Illesszük be prioritási sorba a műveleteket!
  - a egyoperandusú műveletek (-, ++, --, !, ~)
  - a multiplikatív műveletek (*, /, %)
  - az additív műveletek (+, -)
  - bitléptetés (<<, >>)
  - a kisebb-nagyobb relációs műveletek (<=, >=, <, >)
  - az egyenlő-nem egyenlő relációs műveletek (==, !=)
  - bitenkénti 'és' művelet (&)
  - bitenkénti 'kizáró vagy' művelet (^)
  - bitenkénti 'vagy' művelet (|)
  - a logikai 'és' művelet (&&)
  - a logikai 'vagy' művelet (||)
  - a feltételes művelet (? :)
  - értékadó művelet (=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

Típusdefiníció C-ben

- A C nyelvben lehetőségünk van típusok tetszésünk szerinti elnevezésére, azaz típusdefinícióra melyet a

  `typedef`

  kulcsszó vezet be, alakja:

  `typedef típus újnév;`

```
typedef unsigned short int u16;
```

Típusdefiníció C-ben

- A típusdefinícióval élhetünk például akkor, mikor több helyen kell ugyanolyan típusú változót deklarálni, de ez a típus
  - a jövőbeni fejlesztések során esetleg változhat
  - egy bonyolult módon megadható típus, amit nehézkes lenne többször leírni (és a többszöri leírás mellesleg hibaforrás is!)

Felsorolás adattípusok

- Felsorolás adattípus értékhalmaza a típusképzésben felsorolt azonosítók, mint konstans azonosítók által meghatározott értékek.
- A < rendezési reláció a típusképzésben a felsorolás által van definiálva.
- A típusképzésben felsorolt azonosító úgy tekintendő, mintha abban a blokkban deklarált konstans azonosító lenne, amelyik blokkban a típusdefiníció szerepel.

Felsorolás adattípusok

- Felsorolás adattípus definiálása az

  `enum`

  kulcsszó segítségével történik:

  `enum {elem`$_1$`, ..., elem`$_n$**`}`**

```
enum {hetfo, kedd, szerda, csutortok, pentek, szombat,
      vasarnap} nap;              /* változódeklaráció */
```

```
typedef enum {hetfo, kedd, szerda, csutortok, pentek,
            szombat, vasarnap} Het; /* típusdefiníció */
Het nap;                          /* változódeklaráció */
```

Felsorolás adattípusok

- Az első azonosító értéke 0, a többié 1-gyel több az őt megelőzőnél, hacsak mást nem adunk meg.

```
enum {hetfo=1, kedd, szerda, csutortok,
      pentek, szombat, vasarnap} nap;
enum {a=3, b, c, d=2, e, f} ertek;
```

- Felhasználható a nap változó például egy számlált menetű ismétléses vezérlésben

```
for(nap = hetfo; nap <= vasarnap; nap++) {
  /* Ciklusmag ... */
}
```

Felsorolás adattípusok

- A konverzió a felsorolás típus azonosítói és az **`int`** típus között automatikus, sőt, az felsorolás típust a fordító tulajdonképpen az **`int`** típusra vezeti vissza, ezért ennek a típusnak a műveletei megfelelő körültekintéssel használhatóak.
- Azért kell a megfelelő körültekintés, mert például a

  ```
  vasarnap + 1 == 8
  ```

  de a 8 érték már értelmezhetetlen a típus értékeként.

Felsorolás adattípusok

```
typedef enum {hetfo=1, kedd, szerda, csutortok,
      pentek, szombat, vasarnap} Het;
Het nap;

Het KovNap(Het n)
{
  if(n == vasarnap) { return hetfo; }
  else { return n+1; }
}

int main()
{
  for (nap = hetfo; nap <= vasarnap; nap++) {
    /* ??? */
  }
}
```

Logikai adattípus C nyelven

- A C nyelvben nincs logikai (**`bool`**) típus, de azért logikai értékek persze keletkeznek.
- Ha mégis szükségünk lenne logikai érték tárolására, akkor ezt megtehetjük egy **`int`** változóban is.
- Ha egy logikai értéket egy **`int`** típusú változóba tettünk, akkor a logikai hamis érték tárolása után az **`int`** típusú változó értéke 0, az igaz érték tárolása után pedig nem 0. (Sok megvalósításban 1, de ezt nem használhatjuk ki, ha gépfüggetlen programot szeretnénk.)

Logikai adattípus C nyelven

- C-ben nincs logikai konstans (**`false`** és **`true`**), ha mégis szükségünk lenne ezekre, akkor magunk definiálhatjuk őket:

```
#define FALSE   0         /* Boolean típus hamis értéke */
#define TRUE    1         /* Boolean típus igaz értéke  */
```

```
#define TRUE    (1)          /* További lehetséges      */
#define TRUE    (!FALSE)     /* definíciók, de ezek     */
#define TRUE    (!(FALSE))   /* közül egyszerre csak    */
#define TRUE    (0==0)       /* egy használható         */
```

vagy

```
#define TRUE (1==1)
#define FALSE (!TRUE)
```

Logikai adattípus C nyelven

- C-ben bevezetünk egy nem létező **`bool`** típust:

```
typedef enum {false, true} bool;
```

de arra ügyelni kell, hogy egy igaz logikai kifejezés értéke a **`true`** értéktől ilyen definíció esetén (is) eltérhet.

Összetett típusok, típusképzések

- Pointer típus
- Tömb típus
  - Sztringek
- Rekord típus
  - Szorzat-rekord
  - Egyesítési-rekord
- Függvény típus

Összetett adattípusok

- Először a pointer típussal foglalkozunk, mert
  - Ez a C nyelv egy igen hatékony eszköze
  - Használata tömör és hatékony kódot eredményez
  - Bizonyos tevékenységeket csak ezzel lehet megoldani
  - Széles körben használható

POINTEREK

Pointer típus

- Már a K&R is óvatosságra int:
  - "Azt szokták mondani, hogy a mutató, csakúgy, mint a `goto` utasítás, csak arra jó, hogy összezavarja és érthetetlenné tegye a programot. Ez biztos így is van, ha ész nélkül használjuk, hiszen könnyűszerrel gyárthatunk olyan mutatókat, amelyek valamilyen nem várt helyre mutatnak. Kellő önfegyelemmel azonban a mutatókat úgy is alkalmazhatjuk, hogy ezáltal programunk világos és egyszerű legyen."

Pointer típus, dinamikus változók

- Az eddigi tárgyalásunkban szerepelt változók statikusak abban az értelemben, hogy létezésük annak a blokknak a végrehajtásához kötött, amelyben a változó deklarálva lett. A programozónak nincs befolyása a változó létesítésére és megszüntetésére.

- (Ez a fajta statikusság nem tévesztendő össze azzal, amit a **`static`** kulcsszóval érhetünk el a C nyelvben)

Pointer típus, dinamikus változók

- Az olyan változókat, amelyek a blokkok aktivizálásától függetlenül létesíthetők és megszüntethetők, dinamikus változóknak nevezzük.
- Dinamikus változók megvalósításának általános eszköze a pointer típus.

- Egy pointer típusú változó értéke (első megközelítésben) egy meghatározott típusú dinamikus változó.

Pointer típusképzés C-ben

- Pointer típusú változót az alábbi módon deklarálhatunk:

```
típus * változónév;
```

- Például
  - **char** típusú dinamikus változó deklarálása

```
char * pc;
```

  - **unsigned short int** típusú dinamikus változó

```
unsigned short int * pi;
```

Pointer típusképzés C-ben

- Meg kell jegyezni, hogy a **`*`** a változóhoz kötődik, vagyis **`int * p`** kétféle értelmezése közül:
  - A **`*p`** egy **`int`** típusú (dinamikus) változó
  - A **`p`** egy **`int*`** (**`int`**-re mutató pointer) típusú változó

  az első alkalmazásával lehet helyesen értelmezni a következő deklarációkat:
  - **`int *p, r;`**
    - Helyesen: **`*p`** és **`r`** mindketten **`int`** típusú változók
    - Helytelenül: **`p`** és **`r`** mindketten **`int*`** típusú változók
- Ezért szokás a típusmódosító **`*`**-ot szorosan a változóhoz, és nem a típushoz írni

Pointer típusképzés C-ben

- Pointer típust az alábbi módon definiálhatunk:

```
typedef típus *újnév;
```

vagyis egy változódeklarációhoz hasonlóan, csak a változónév helyett az új típus neve szerepel.

- Például

```
typedef unsigned long int *ulip;
ulip p;
```

Hivatkozás, változó, érték

- Első megközelítésben tehát egy pointer értéke egy dinamikus változó.
- Az eddigiek során lényegében azonosítottuk a változóhivatkozást és a hivatkozott változót.
- A dinamikus változók megértéséhez viszont világosan különbséget kell tennünk az alábbi három fogalom között:
  - változóhivatkozás
  - hivatkozott változó
  - változó értéke

Hivatkozás, változó, érték

- A változóhivatkozás szintaktikus egység, tehát meghatározott formai szabályok szerint képzett jelsorozat egy adott programnyelven.

- A változó a program futása során a program által lefoglalt memóriaterület egy része, amelyen egy adott (elemi vagy összetett) típusú érték tárolódik.

Hivatkozás, változó, érték

- Különböző változóhivatkozások hivatkozhatnak ugyanarra a változóra, illetve ugyanaz a változóhivatkozás a végrehajtás különböző időpontjaiban különböző változókra hivatkozhat.

- Egy változóhivatkozáshoz nem biztos, hogy egy adott időben tartozik hivatkozott változó.

Pointer típus műveletei

- Tegyük fel, hogy a műveletek definiálásában szereplő **`p`**, **`q`** változók deklarációja a következő:

```
typedef E *PE;
PE p, q;
```

Pointer típus műveletei

- Új dinamikus változó létesítése

  `p=malloc(sizeof(E))`

  – A **`malloc(S)`** függvény lefoglal egy **`S`** méretű memóriaterületet a program számára. A **`sizeof(E)`** megadja, hogy egy **`E`** típusú változó mekkora helyet igényel a memóriában. A **`malloc(sizeof(E))`** hatására tehát létrejön egy új **`E`** típusú érték tárolására (is) alkalmas változó, és ez a változó lesz a **`p`** értéke.

Pointer típus műveletei

- Pointer dereferencia

  `*p`

  – A **`*`** művelet segítségével érhetjük el a **`p`** értékét vagyis a dinamikus változót. A **`*p`** változóhivatkozás a **`p`** értékére, vagyis a dinamikus változóra hivatkozik, tehát a **`*p`** értéke a dinamikus változó értéke lesz.

Pointer típus műveletei

- Dinamikus változó törlése

  `free(p)`

  – A művelet hatására a **`p`**-hez tartozó memóriaterület felszabadul ezáltal a dinamikus változó megszűnik. A művelet végrehajtása után a **`p`** pointerhez nem tartozik érvényes változó, ezért a **`*p`** változóhivatkozás végrehajtása jobb esetben azonnali futási hibát eredményez. (Rosszabb esetben pedig olyan lappangó hibát, aminek az eredménye a program egy teljesen más pontján jelenik meg.)

Pointer típusműveletei

- A statikus változóhivatkozáshoz tartozó változó a blokk végrehajtásának megkezdésétől a befejezéséig létezik.
- Dinamikus változóhivatkozáshoz tartozó változók a pointer típus műveleteivel hozhatók létre és szüntethetők meg.

Pointer típus műveletei

- A dimamikus változó létrehozására tehát a **`malloc()`**, megszűntetésére a **`free()`** függvény szolgál:

```
int *p;
 p  = malloc(sizeof(int));
*p  = 3;
*p += 6;
free(p);
```

Változó	Érték
p	*p
*p	9

Pointer típus műveletei

- Null-pointer, konstans érték

  `NULL`

  – Minden pointer adattípus értékhalmazának eleme a **`NULL`** érték. Ha a **`p`** változó **`NULL`**, akkor biztos, hogy a **`*p`** változóhivatkozáshoz nem tartozik dinamikus változó.

- Értékadás

  `q=p`

  – A művelet végrehajtása után a **`*q`** is ugyanarra a dinamikus változóra fog hivatkozni, mint **`*p`**.

Pointer típus műveletei

- A dimamikus változó létrehozására tehát a **`malloc()`**, megszűntetésére a **`free()`** függvény szolgál:

```
int *p = NULL, *q = NULL;
 p  = malloc(sizeof(int));
*p  = 3;
 q  = p;
*q += 6;
free(q);
```

Változó	Érték
p	din.v.
q	din.v.
din.v.	9

Pointer típus

- A pointer típus az értékadás és a dereferencia segítségével lehetővé teszi dinamikus változók összekapcsolását, ezáltal komplex adatszerkezetek létrehozását. Erre a későbbiekben több példát is láthatunk majd.

Pointer típus műveletei

- Egyenlőség relációs művelet

  `p == q`

  – A művelet értéke akkor és csak akkor igaz, **`p`** és **`q`** értéke megegyezik, vagyis az **`*p`** és **`*q`** ugyanarra a változóra hivatkozik, vagy mindkettő **`NULL`**.

- Nem egyenlő relációs művelet

  `p != q`

  – A művelet értéke akkor és csak akkor igaz, ha **`p`** és **`q`** értéke nem egyezik meg, vagyis az **`*p`** és **`*q`** különböző változókra hivatkoznak, vagy az egyik **`NULL`**.

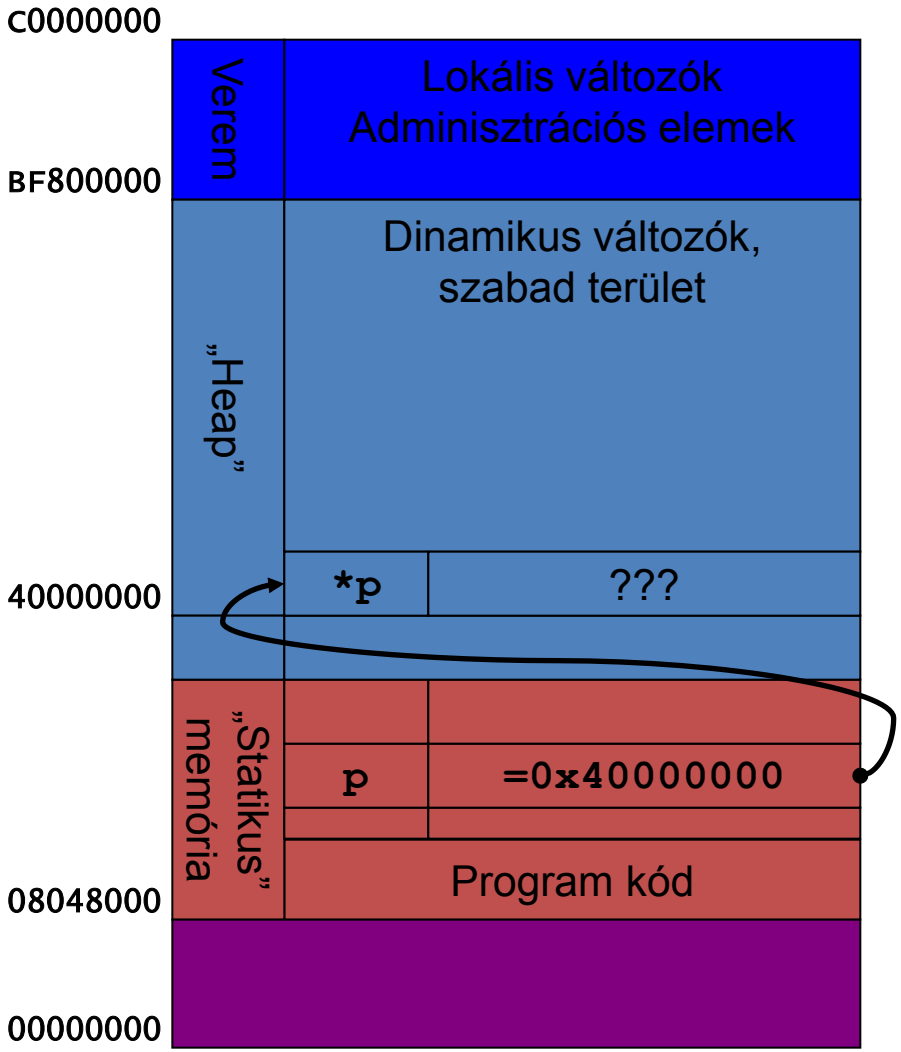
Pointer típus

- A pointer típus másik megközelítése a változó fogalmából indul ki.
- A gépi megvalósítást tekintve minden változóhoz tartozik egy memóriamező, ezért a `p` pointer típusú változó értéke egy másik változóhoz tartozó memóriamező címeként értelmezhető.

- Ez az értelmezés egyébként egybeesik a pointer típus megvalósításával.

Pointer típus megvalósítása

- Logikailag minden programnak saját memória-tartománya van (OS biztosítja)
- Pointer típusú változó a hozzá tartozó dinamikus változóhoz foglalt memóriamező kezdőcímét tartalmazza.



Pointer típus műveletei

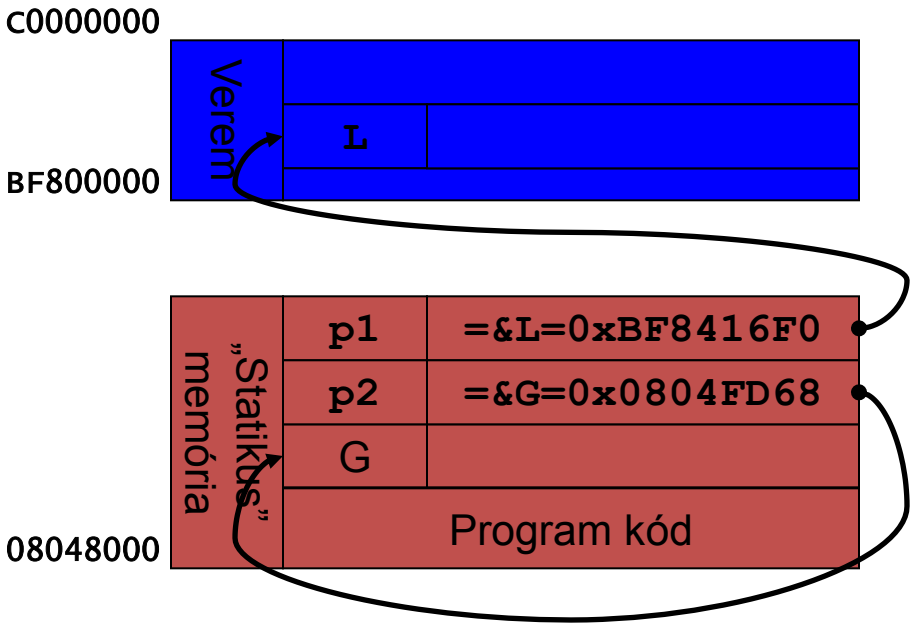
- A címképző művelet

  `p = &i`

  – A művelet meghatározza egy változóhoz tartozó memória mező címét. Ha egy **`p`** pointer típusú változó értéke az **`i`** változóhoz tartozó memória címe, akkor azt mondjuk, hogy a **`p i`**-re mutat.

Pointer típus műveletei

- Az **&** művelet tehát egy változó memóriacímét adja vissza, így egy pointer értéke akár egy globális vagy lokális statikus változó is lehet.
- Arra azért figyeljünk, hogy a lokális változók csak addig léteznek, míg az adott blokkból ki nem lépünk.



Pointer típus műveletei

- Az **&** és a ***** jobbasszociatív műveletek és magas precedenciájúak.

```
int i,j;
   int *p;  /* a *p memória mezőn int egy eleme tárolható */
            /* lehetne együtt is deklarálni: int *p,i,j;  */
    p = &i;         /* p i-re mutat */
    j = *p;         /* hatása ua., mint j = i;  */
   *p = 2;          /* hatása ua., mint i = 2; */
    j = *p + 1;     /* hatása ua., mint j = i + 1; */
   *p += 1;         /* hatása ua., mint i += 1; */
   ++*p;            /* hatása ua., mint ++i; */
   (*p)++;          /* hatása ua., mint i++; */
```

Pointer típus

- A prioritási előírás csökkenő sorrendben
  - a egyoperandusú műveletek (-, ++, --, !, ~, &, *)
  - a multiplikatív műveletek (*, /, %)
  - az additív műveletek (+, -)
  - bitléptetés (<<, >>)
  - a kisebb-nagyobb relációs műveletek (<=, >=, <, >)
  - az egyenlő-nem egyenlő relációs műveletek (==, !=)
  - bitenkénti 'és' művelet (&)
  - bitenkénti 'kizáró vagy' művelet (^)
  - bitenkénti 'vagy' művelet (|)
  - a logikai 'és' művelet (&&)
  - a logikai 'vagy' művelet (||)
  - a feltételes művelet (? :)
  - értékadó művelet (=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)
  - szekvencia művelet (,)

void *

- A **void** típust már ismerjük: ez az amelynek az értékkészlete 0 elemű. Miért van szükség **void** típusra mutató pointerre vagy ilyen típusú dinamikus változóra?
- A **void*** egy speciális, úgynevezett típustalan pointer. Az ilyen típusú pointerek „csak” memóriacímek tárolására alkalmasak, a dereferencia művelet alkalmazása rájuk értelmetlen. Viszont az értékadás mindkét oldalán használhatóak.

Függvény argumentumok

- A függvényműveletnél a paraméterek kezelése érték szerint történik, amely csak bemenő módú paraméter használatát teszi lehetővé.
- Ha kimenő módú paraméterre is szükségünk van, akkor ennek kezelését nekünk kell megoldani pointer segítségével.
- A csere függvényművelet feladata az argumentumok értékének megcserélése.

Függvény argumentumok

- Ha csak így deklaráljuk a csere függvényt

```
csere (int x, int y)
{
   int m;
   m = x;
   x = y;
   y = m;
}
```

akkor a **`csere(a, b);`** nem végzi el a cserét, hiszen csak az **`a`** és **`b`** változó értékét kapta meg és ezeket használta a lokális **`x`**, **`y`**, **`m`** változóknál.

Függvény argumentumok

- Át kell tehát adni az a és b változók címét, hogy az értékük ténylegesen megcserélhető legyen:

```
csere (&a, &b);
```

- Ekkor a csere függvényművelet deklarációja:

```
csere (int *px, int *py)
{
   int m;
     m = *px;
   *px = *py;
   *py = m;
}
```

Függvény argumentumok

- Ha nem használunk globális változókat, akkor kimenő módú argumentum kezelésére kielégítő megoldást ad a következő séma is:
  - A lokális változóban lévő értéket a **`return`** utasítás előtt adjuk át az aktuális paraméternek

```
csinalValamit (int *px, int *py) {
  int x, y;

  ...
  *px = x; *py = y;
  return;
}
```

Függvény argumentumok

- Ha nem használunk globális változókat, akkor be- és kimenő módú argumentum kezelésére kielégítő megoldást ad a következő séma is:
  - Az aktuális paraméterek értékét lokális változóba tesszük, majd ezek értékit a **`return`** utasítás előtt visszaadjuk aktuális paraméternek

```
csinalValamit (int *px, int *py) {
  int x, y;
  x = *px; y = *py;
  ...
  *px = x; *py = y;
  return;
}
```

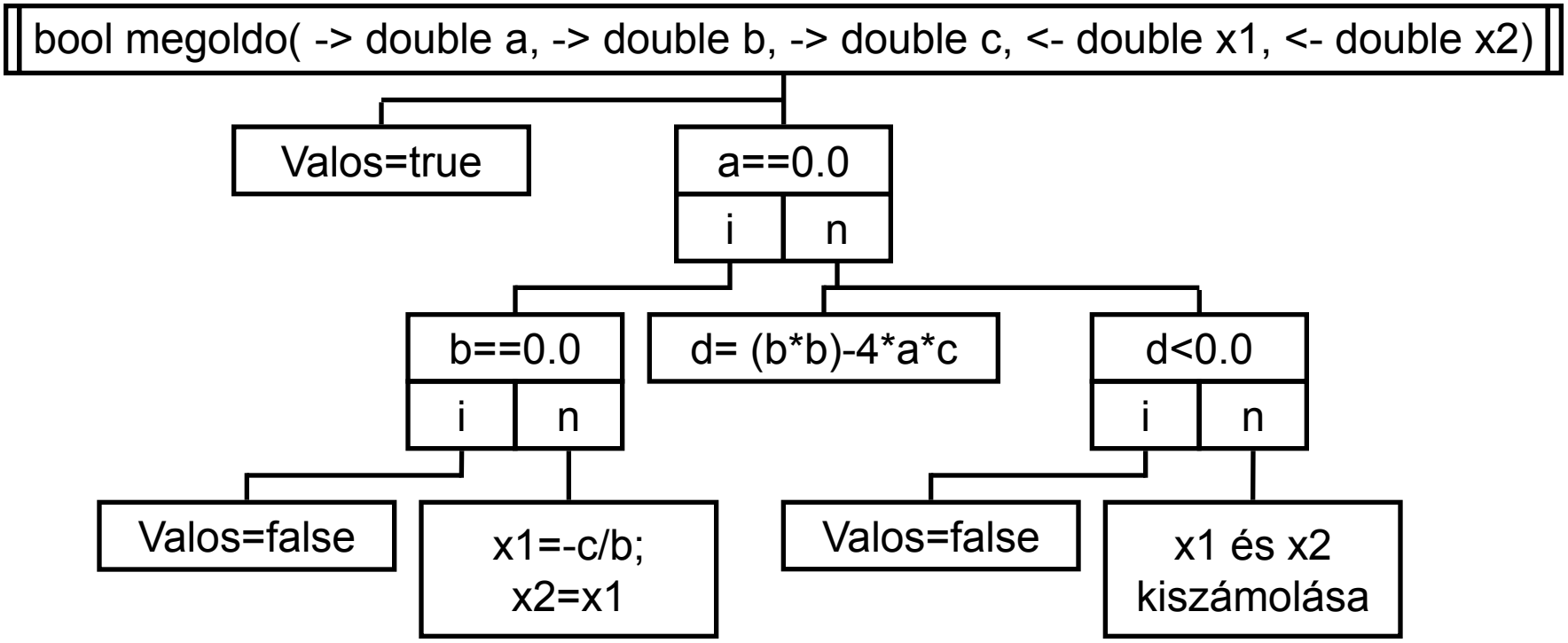
Másodfokú egyenlet

- Problémafelvetés:
  - Másodfokú egyenlet valós gyökeinek meghatározása.
- Specifikáció:
  - Input
    - a,b,c valós szám
  - Output
    - x,y valós szám, a másodfokú egyenlet gyökei, ha vannak, különben egy szöveg, hogy nincs valós gyök.

Másodfokú egyenlet

- Algoritmustervezés:
  - A probléma megoldása olyan függvényművelettel adható meg, amelynek három bemenő paramétere van, az egyenlet együtthatói, és két kimenő paramétere van, a két valós gyök, továbbá a függvény logikai értéket ad vissza, amely akkor és csak akkor lesz igaz, ha van valós gyök.
- A közismert megoldóképletet használjuk.

Másodfokú egyenlet



Másodfokú egyenlet

```
/* Másodfokú egyenlet valós gyökeinek meghatározása a
 * megoldo függvénnyel.
 * 1998. március 31. Dévényi Károly, devenyi@inf.u-szeged.hu
 */

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

typedef enum {false, true} boolean;
```

>>>

Másodfokú egyenlet

```
boolean megoldo(double a, double b, double c,  /* együtthatók */
                double *x1, double *x2)              /* gyökök */
{
  double d;                                      /* a diszkrimináns */
  boolean valos;                                  /* van-e megoldás */
  valos = true;
  if (a == 0.0) {
      if (b == 0.0) {                      /* az egyenlet elfajuló */
          valos = false;
      } else {
          *x1 = -(c / b);
          *x2 = *x1;
      }
  } else {
```

Másodfokú egyenlet

```
        d = b * b - 4.0 * a * c;
        if (d < 0.0) {                          /* nincs valós gyöke */
            valos = false;
        } else {
            *x1 = (-b + sqrt(d)) / (2.0 * a);
            *x2 = (-b - sqrt(d)) / (2.0 * a);
                                    /* a gyökök pontosabb kiszámolása */
            if (fabs(*x1) > fabs(*x2)) {
                *x2 = c / (*x1 * a);
            } else if (*x2 != 0) {
                *x1 = c / (*x2 * a);
            }
        }
    }
    return valos;
}
```

>>>

Másodfokú egyenlet

```
main()
{
  double a, b, c, x, y;                    /* a főprogram változói */
  printf("Kérem az első egyenlet együtthatóit!\n");
  scanf("%lg%lg%lg%*[^\n]", &a, &b, &c);
  getchar();
  if (megoldo(a, b, c, &x, &y)) {
      printf("Az egyenlet gyökei: %20.10f és %20.10f\n", x, y);
  } else {
      printf("Az egyenletnek nincs valós megoldása!\n");
  }
```

Másodfokú egyenlet

```
  printf("Kérem a második egyenlet együtthatóit!\n");
  scanf("%lg%lg%lg%*[^\n]", &a, &b, &c);
  getchar();
  if (megoldo(a, b, c, &x, &y)) {
      printf("Az egyenlet gyökei: %20.10f és %20.10f\n", x, y);
  } else {
      printf("Az egyenletnek nincs valós megoldása!\n");
  }
}
```

Másodfokú egyenlet

- Az x1 és x2 kimenő paraméterek, ezért átalakítjuk a deklarációt.
- Ezt megtehetjük, hiszen nem használunk globális változókat.

Másodfokú egyenlet

```
boolean megoldo(double a, double b, double c,  /* együtthatók */
                double *x1, double *x2)              /* gyökök */
{
  double d;                                    /* a diszkrimináns */
  boolean valos;                                /* van-e megoldás */
  double mx1, mx2;           /* munkaváltozók x1 és x2 helyett */
  valos = true;
  if (a == 0.0) {
      if (b == 0.0) {                  /* az egyenlet elfajuló */
          valos = false;
      } else {
          mx1 = -(c / b);
          mx2 = mx1;
      }
  } else {
     d = b * b - 4.0 * a * c;
```

>>>

Másodfokú egyenlet

```
        if (d < 0.0) {                          /* nincs valós gyöke */
            valos = false;
        } else {
            mx1 = (-b + sqrt(d)) / (2.0 * a);
            mx2 = (-b - sqrt(d)) / (2.0 * a);
                                  /* a gyökök pontosabb kiszámolása */
            if (fabs(mx1) > fabs(mx2)) {
                mx2 = c / (mx1 * a);
            } else if (mx2 != 0) {
                mx1 = c / (mx2 * a);
            }
        }
    }
    *x1 = mx1;
    *x2 = mx2;
    return valos;
}
```

TÖMBÖK

Tömb típus

- Algoritmusok tervezésekor gyakran előfordul, hogy adatok sorozatával kell dolgozni, vagy mert az input adatok sorozatot alkotnak, vagy mert a feladat megoldásához kell.
- Tegyük fel, hogy a sorozat rögzített elemszámú (n) és mindegyik komponensük egy megadott (elemi vagy összetett) típusból (E) való érték.

Tömb típus

- Ekkor tehát egy olyan összetett adathalmazzal van dolgunk, amelynek egy eleme

  $A = (a_0, \ldots, a_{n-1})$

  ahol a_i eleme E (i=0, ... ,n-1).
- Ha az ilyen sorozatokon a következő műveleteket értelmezzük, akkor egy (absztrakt) adattípushoz jutunk, amit Tömb típusnak nevezünk.
- Jelöljük a Tömb típust T-vel, a 0..n-1 intervallumot pedig I-vel.

Tömb típus műveletei

- Kiolvas(-> A:T; -> i:I; <- x:E)
  - Adott i eleme I -re az A sorozat i. komponensének kiolvasása adott x, E típusú változóba.
- Módosít(<-> A:T; -> i:I; -> y:E)
  - Adott i eleme I -re az A sorozat i. komponensének módosítása adott y, E típusú értékre.
- X=Y
  - Értékadó művelet.

Riadólánc

- Problémafelvetés:
  - Adott n számú embert tartalmazó közösség, akik riadóláncot akarnak alkotni. A közösség minden tagjára meghatározott, hogy kit értesítsen. Eldöntendő, hogy egy ilyen hozzárendelés valóban riadóláncot alkot-e?

Riadólánc

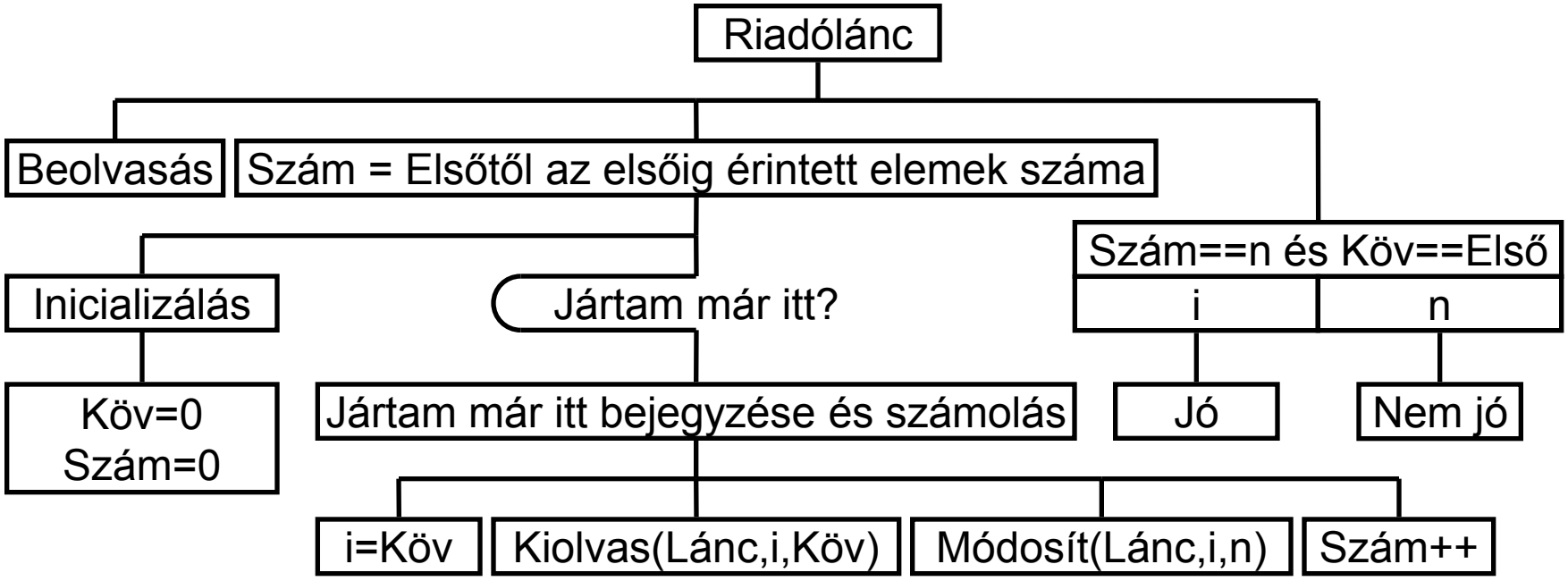
- Specifikáció:
  - Azonosítsuk a közösség tagjait az [0..n-1] intervallum elemeivel.
  - Input
    - Egy n elemű számsorozat.
  - Output
    - Igen/Nem tetszőleges szövegkörnyezetben.

Riadólánc

- Algoritmustervezés:
  - A riadólánc egy Lánc nevű számsorozattal adható meg, amelynek i. eleme annak az embernek a sorszáma, akit az i-nek értesíteni kell.
  - A Lánc sorozat akkor és csak akkor valódi riadólánc, ha bármelyik elemétől elindulva a Lánc szerinti hozzárendelést követve visszajutunk i-be úgy, hogy közben minden elemet érintettünk.
  - Ezt azonban elég egy tetszőleges elemtől kezdve kipróbálni.

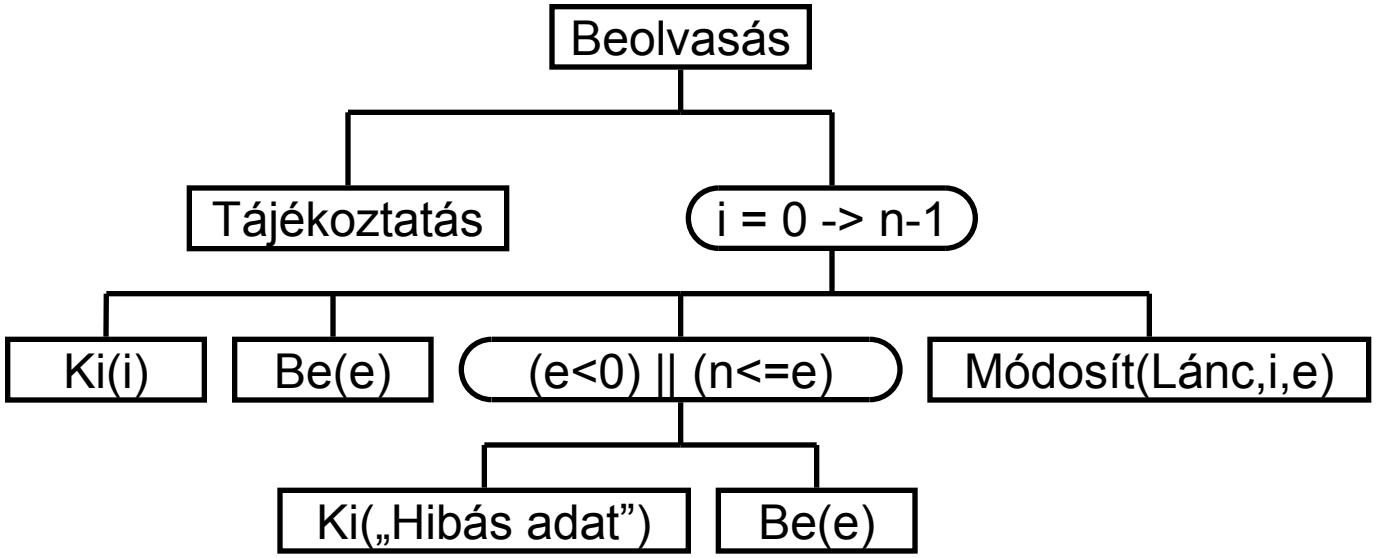
Riadólánc

- Struktúradiagramm:



Riadólánc

- Struktúradiagramm:



Tömb típusképzés C-ben

- Tömb típusú változót az alábbi módon deklarálhatunk:

```
típus változónév[elemszám];
```

- Például
  - **char** típusú elemekből álló 5 elemű tömb deklarálása

```
char ct[5];
```

  - **unsigned short int** típusú 20 elemű tömb

```
unsigned short int it[20];
```

Tömb típusképzés C-ben

- Tömb típust az alábbi módon definiálhatunk:

```
typedef típus újnév[elemszám];
```

vagyis egy változódeklarációhoz hasonlóan, csak a változónév helyett az új típus neve szerepel.

- Például

```
typedef int tomb20[20];
tomb20 t;
```

Tömb típus műveletei C-ben

- A Kiolvas és a Módosít műveletek megvalósítása a tömbelem-hivatkozással történik.
- A tömbelem-hivatkozásra a

  `[ ]`

  zárójelpárt használjuk
- Felfoghatjuk a `[ ]`-t az indexelés művelet operátorának is, aminek az eddigi legmagasabb precedenciával kell rendelkeznie és balasszociatív.

Tömb típus műveletei C-ben

- A prioritási előírás csökkenő sorrendben
  - elemkiválasztások ([])
  - a egyoperandusú műveletek (-, ++, --, !, ~, &, *)
  - a multiplikatív műveletek (*, /, %)
  - az additív műveletek (+, -)
  - bitléptetés (<<, >>)
  - a kisebb-nagyobb relációs műveletek (<=, >=, <, >)
  - az egyenlő-nem egyenlő relációs műveletek (==, !=)
  - bitenkénti 'és' művelet (&)
  - bitenkénti 'kizáró vagy' művelet (^)
  - bitenkénti 'vagy' művelet (|)
  - a logikai 'és' művelet (&&)
  - a logikai 'vagy' művelet (||)
  - a feltételes művelet (? :)
  - értékadó művelet (=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)
  - szekvencia művelet (,)

Tömb típus műveletei C-ben

- Kiolvas(a, i, x)

  ```
  x = a[i]
  ```

- Módosít(a, i, x)

  ```
  a[i] = x
  ```

- Ha az **a** és a **b** is tömb típusú változó, akkor az

  ```
  a = b
  ```

  értékadás nem megengedett, mert a baloldalon nem változóhivatkozás áll. Erről később még bővebben lesz szó.

Tömb típus C-ben

- Meg kell jegyezni, hogy a C nyelvben a tömb indexelése minden esetben 0-val kezdődik, azaz egy **`int t[20]`** deklaráció esetén **`t[0],t[1],…,t[19]`** lesznek a tömb elemei.
- A C nyelvben viszont nincs indexellenőrzés, azaz egy **`int t[20]`** deklaráció esetén hivatkozni lehet például a **`t[20]`** vagy **`t[-1]`** elemekre is, ez azonban nagyon csúnya futási hibákat eredményezhet.

Riadólánc

```
/* Adott n számú embert tartalmazó közösség. Eldöntendő,
 * hogy riadóláncot alkotnak-e?
 * 2006. augusztus 9. Gergely Tamás, gertom@inf.u-szeged.hu
 */

#include <stdio.h>

#define N    8              /* a sorozat elemeinek száma */
#define ELSO 0     /* az első vizsgálandó elem sorszáma */

main ()
{
   int A[N];                  /* a sorozatot tároló tömb */
   int e,i;                             /* munkaváltozók */
   int kov;                   /* a következő vizsgálandó */
   int szam;                              /* számláló */
```

>>>

Riadólánc

```
printf("Kérem a %d elemű sorozatot, amelyről ", N);
printf("eldöntöm, hogy riadólánc-e!\n");

for(i = 0; i < N; ++i) {                    /* beolvasás */
   printf("%d. kit értesít ? ", i);
   scanf("%d%*[^\n]", &e);
   getchar();
   while((e < 0) || (N <= e)) {
      printf("Hibás adat!\nKérem újra: ");
      scanf("%d%*[^\n]", &e);
      getchar();
   }
   A[i] = e;
}
```

Riadólánc

```
   kov = ELSO; szam = 0;                  /* inicializásás */
   do {
      i    = kov;
      kov  = A[i];                          /* továbblépés */
      A[i] = N;              /* jártam már itt bejegyzése */
      szam++;
   } while (A[kov] != N);                /* jártam már itt? */

   if ((szam == N) && (kov == ELSO)) {
      printf("A számsorozat riadólánc.\n");
   } else {
      printf("A számsorozat nem riadólánc.\n");
   }
}
```

Általános absztrakt tömb típus

- Legyen E tetszőleges típus
- Legyenek I_1, ... ,I_k tetszőleges sorrendi típusok
- Legyen $I=I_1 \times ... \times I_k = \{ (i_1, ... ,i_k) \mid i_1 \text{ in } I_1, ... ,i_k \text{ in } I_k \}$, tehát az I_1, ... ,I_k halmazok direktszorzata.
- Képezhetjük azt a T=Tömb(I_1, ... ,I_k,E) új típust, amelynek értékhalmaza az I-ből E-be való függvények halmaza, azaz:

  { A | A : I --> E }
- A k-t a tömb dimenziójának nevezzük.

Általános absztrakt tömb típus

- A T=Tömb(I_1, ... ,I_k,E) k dimenziós tömb típus az alábbi három művelettel rendelkezik
  - Kiolvas(-> A:T; -> i_1:I_1; ... ; -> i_k:I_k; <- x:E)
    - A művelet végrehajtása után x==A(i_1, ... ,i_k).
  - Módosít(<-> A:T; -> i_1:I_1; ... ; -> i_k:I_k; -> x:E)
    - A művelet végrehajtása után A(i_1, ... ,i_k)==x és a többi argumentumokra A értéke nem változik.
  - Az X=Y értékadás értelmezett, ha az X és Y változók típusa T

Általános absztrakt tömb típus

- Vegyük észre, hogy a T=Tömb(I_1, ... ,I_k,E) típus felfogható úgy, mint T=Tömb(I_1, T2), ahol T2=Tömb(I_2, ... ,I_k,E).
- Továbbá az sem jelent megkötést, hogy minden egyes I_j intervallum bal végpontja a 0 legyen, hiszen az [n..m] intervallum elemei egyszerűen transzformálhatók a [0..(m-n)] intervallumra.
- Ezek alapján a C nyelven már létre tudunk hozni többdimenziós tömböket is.

Általános tömb típus C nyelven

- Legyen

  $T=\text{Tömb}(I_1, \ldots, I_k, E)$

  ahol

  I_j a $[0..N_j-1]$ intervallum

- Ennek a típusnak a definíciója C nyelven:

```
typedef E T[N1][N2]...[Nk];
```

Általános tömb típus C nyelven

- Például
  - típusdefiníciók

```
typedef int tomb[100];
typedef double matrix[10][10];
typedef char szoveg[21];
```

  - változódeklarációk

```
matrix m;     /* az előző típussal*/
char s[21];
int vector[20];
```

Általános tömb típus C nyelven

- A tömbelem-hivatkozás itt is a **`[]`** zárójelpárral történik úgy, hogy minden indexet külön zárójelek közé teszünk:

  `m[5][7]`

- Ez tulajdonképpen nem más, mint a **`[]`** operátor többszöri alkalmazása, hiszen:
  - **`m`** egy tömb, aminek egy eleme tömb, aminek egy eleme **`double`** változó
  - **`m[5]`** egy tömb, aminek egy eleme **`double`** változó
  - **`m[5][7]`** egy **`double`** változó

Tömb típus műveletei C nyelven

- A Kiolvas és a Módosít műveletek megvalósítása most is a tömbelem-hivatkozással történik.
- Kiolvas(a,i_1, ... ,i_k,x)

```
x = a[i1]...[ik]
```

- Módosít(a,i_1, ... ,i_k,x)

```
a[i1]...[ik] = x
```

- Ha az **a** és **b** is **T** tömb típusú változó, akkor az **a = b** értékadás nem megengedett, mert a baloldalon nem változóhivatkozás áll. Erről később még bővebben lesz szó.

Tömb gépi szintű megvalósítása

- A lehetséges gépi megvalósítás vizsgálatánál abból kell kiindulni, hogy a memória lineáris szerkezetű.
- Tömb típusú változó számára történő helyfoglalás azt jelenti, hogy minden tömbelem, mint változó számára memóriát kell foglalni.
- Feltehetjük, hogy egy adott tömb változóhoz a tömbelemek számára foglalt tárterület összefüggő mezőt alkot.

Tömb gépi szintű megvalósítása

- A megvalósítás tehát azt jelenti, hogy a lefoglalt memóriaterület kezdőcíme és az i_1, ... ,i_k indexkifejezések értékéből ki kell számítani az A[i_1]...[i_k] tömbelem Cím(A[i_1]...[i_k]) címét. Ezt a hozzárendelést az A tömbváltozóhoz tartozó címfüggvénynek nevezzük.
- Nyilvánvaló, hogy a Cím függvény felírható t_0+TCF(i_1,...,i_k) alakban, ahol t_0 az A változóhoz foglalt memóriamező kezdőcíme, a TCF függvény pedig a tömb típus által meghatározott és tömb-címfüggvénynek nevezzük.

Tömb gépi szintű megvalósítása

- Olyan TCF függvényt keresünk, amely egyszerűen, gyorsan kiszámítható i_1, ... ,i_k függvényében, tehát $c_0+c_1*i_1+...+c_k*i_k$ lineáris alakban.
- Legyen a szóbanforgó tömb típus definíciója a következő:

```
typedef E T[N1]...[Nk];
T A;
```

ahol N_1, ... ,N_k konstansok.

Tömb gépi szintű megvalósítása

- Legyen h=sizeof(E). Ekkor az A változó számára foglalandó memória mérete $h*(N_1*...*N_k)$.
- Alkalmas TCF függvényt kaphatunk úgy, hogy az $(i_1,...,i_k)$ indexértékek halmazán definiálunk egy < lineáris rendezési relációt és a Sorsz$(i_1,...,i_k)$ sorszámfüggvénnyel képezzük a h*Sorsz$(i_1,...,i_k)$ kifejezést, ahol Sorsz$(i_1,...,i_k)$ a rendezésben az $(i_1,...,i_k)$ elemet megelőző elemek száma.
- A leggyakrabban rendezésként az úgynevezett lexikografikus rendezést használják.

Tömb gépi szintű megvalósítása

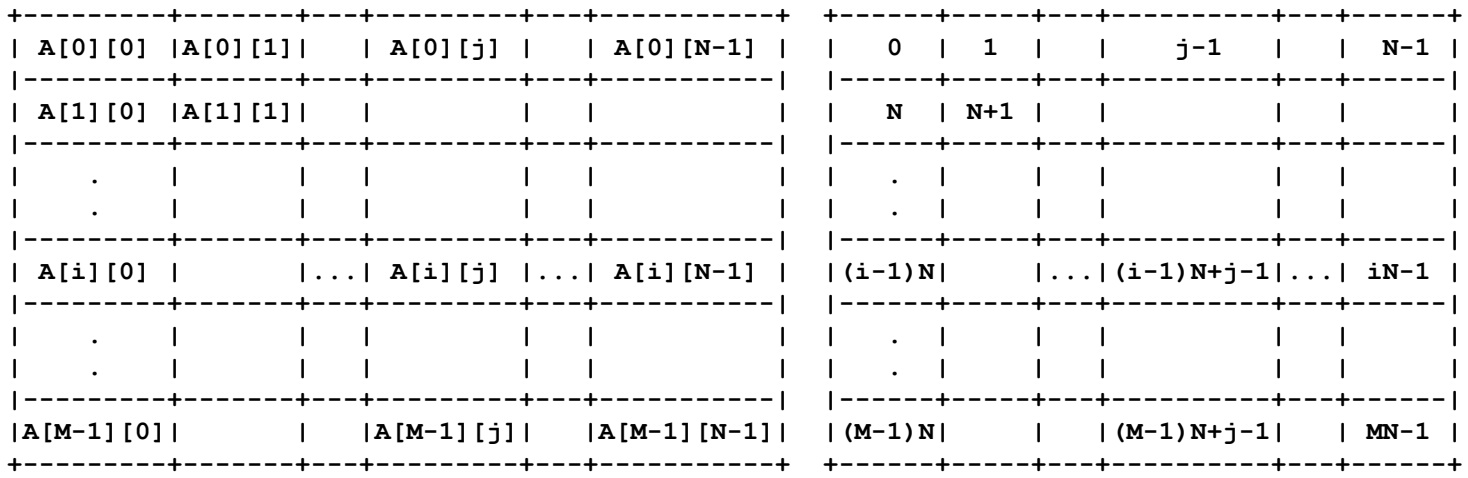
- $(i_1, \ldots, i_k) <_{lex} (j_1, \ldots, j_k)$ akkor és csak akkor, ha a legkisebb u indexre, amelyre $i_u != j_u$, teljesül az $i_u < j_u$.
- Megmutatható, hogy ekkor a sorszámfüggvény lineáris, pontosabban

  $$Sorsz(i_1, \ldots, i_k) = d_0 + d_1 * i_1 + \ldots + d_k * i_k,$$

  ahol d_i konstansok a program fordításakor kiszámíthatók, így az $A[i_1]\ldots[i_k]$ tömbelem címének kiszámításakor nem kell újra számolni ezen értékeket.

Tömb gépi szintű megvalósítása

- Kétdimenziós esetben a lexikografikus elrendezést sorfolytonos elrendezésnek is nevezik, mert egy A tömb elemei ekkor táblázatban így rendezhetők el:



Tömb típus megvalósítása

- A C is a lexikografikus rendezés szerint számítja a címfüggvényt.
- Tömbök használata nagy körültekintést igényel, mert a program végrehajtása közben nincs indexhatár-ellenőrzés, így

```
double m[10][10];
```

deklaráció esetén

```
m[0][10];
```

ugyanaz mint

```
m[1][0];
```

Tömb típus megvalósítása

- Példa:

```
int i, j, a[3][3];
for(i=0; i<9; ++i)
   a[0][i]=i;
for(i=0; i<3; ++i) {
   for(j=0; j<3; ++j)
      printf("%d ", a[i][j]);
   printf("\n");
}
```

- Az output:

```
0 1 2
3 4 5
6 7 8
```

Ciklikus

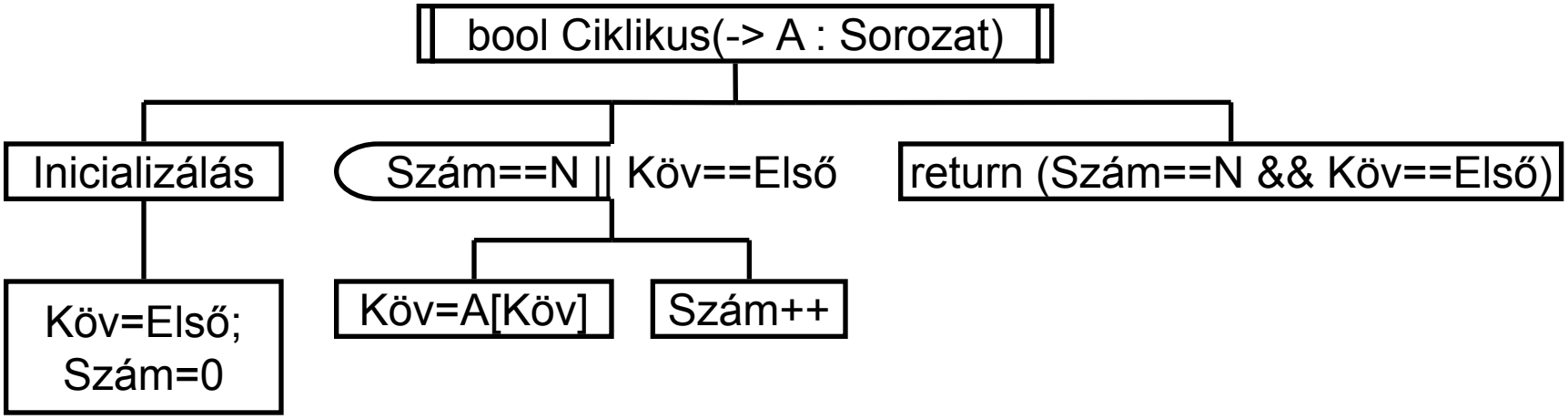
- A Riadólánc algoritmusban a bemenő adatot tároló tömb az algoritmus végrehajtása során módosul. Ez elkerülendő akkor, ha a bemenő adaton más műveletet is kell végeznünk.
- Ezt a követelményt is kielégítő megoldást láthatunk a Ciklikus nevű függvényben.
- Az elnevezés arra utal, hogy az ilyen sorozatokat a matematikában ciklikus permutációknak nevezik.

Ciklikus

- Konvenció:
  - Probléma megoldását program helyett kifejezhetjük eljárás vagy függvény formájában is, ekkor azonban fel kell tüntetni az eljárás/függvény használatához szükséges globális programegységeket.

Ciklikus

- Struktúradiagramm:



Tömb mint paraméter

- Egy függvény paramétere lehet tömb típusú is. Ilyen esetben azonban csak a tömb címe, vagyis egy pointer kerül átadásra, így a tömb elemein végzett bármely módosítás kihat az eredeti tömbre.

Ciklikus

```
/*  Globális programegységek a Ciklikus függvényhez   */
#define N    ???          /* a sorozat elemeinek száma */
#define ELSO 1
typedef int bool;

bool ciklikus(int A[] /* A paraméter egy tömb */)
/* Eldönti, hogy az A sorozat ciklikus permutáció-e?
 * 2005. október 13.
 * Gergely Tamás, gertom@inf.u-szeged.hu
 */
{
   int kov = ELSO, szam = 0; /* következő és számláló */
   do {
      kov = A[kov]; szam++;             /* továbblépés */
   } while((Kov != ELSO) && (Szam != N));
   return (kov == ELSO) && (Szam == N);
}
```

Tömb mint paraméter

- Mivel C-ben nincs indexhatár-ellenőrzés, és a paraméterként átadott tömbnek csak a címét kapja meg a függvény, ezért ha a függvény T paramétertípusa E típusú elemekből álló egydimenziós tömb, ennek a pontos méretét a függvény deklarációjában nem kell megadni.

Tömb mint paraméter

- Az E típus méretét viszont pontosan ismerni kell, tehát ha a T többdimenziós tömb típus, azaz E is legalább egydimenziós tömb típus, akkor E minden dimenziójának a méretét pontosan fel kell tüntetni, azaz csak T legelső dimenziójának mérete hagyható el. Pl:

```
int fgv(int tomb[][10][3]);
```

- Ez az egyes elemek címének pontos kiszámítása végett szükséges.

Egy komplexebb adatszerkezet

- A gráf megvalósítható tömbök segítségével többféle módon is, attól függően, hogy milyen műveleteket fogunk végezni rajta:
  - Két adott pont között van-e él?
  - Adott pontból hány él vezet ki, és hová?
  - Kell-e törölni éleket vagy elég ha a bemenő élek számát csökkentjük?

	1	2	3	4
1	0	1	0	1
2	0	0	0	1
3	1	1	0	0
4	0	0	1	1

	Ki				Be
1	2	2	4		1
2	1	4			2
3	2	1	2		1
4	2	3	4		3

SZTRINGEK

Sztringek C-ben

- C-ben, karaktersorozatot egyszerűen egy karakteres tömbbel készítünk.

```
char str[h];
```

- A **`char str[h]`** változó számára **`h`** bájt foglalódik és maximum **`h-1`** karakter hosszú szöveg tárolható benne.
- A szöveg maximális hosszára (a fizikai korlátokon kívül) nincs korlátozás.

Sztringek C-ben

- Az **`str`** szöveg **`i`**. karakterére az

  `str[i-1]`

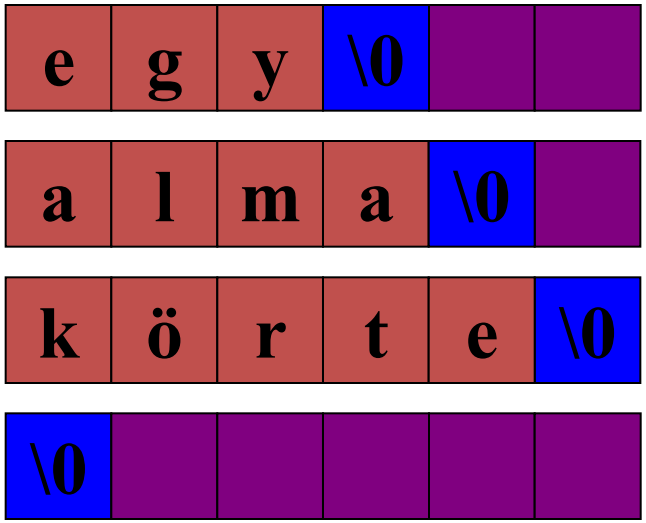
  változóhivatkozással hivatkozhatunk.
- A szöveg végét a szöveghez tartozó utolsó karakter után elhelyezett **`'\0'`** karakter jelzi.
- Ezért van, hogy egy **`h`** hosszúságú karaktertömbben maximum **`h-1`** értékes karaktert, ezáltal maximum **`h-1`** karakter hosszú szöveget tárolhatunk.

Sztringek C-ben

- Legyen

```
char str[6];
```

- Ekkor az **"egy"**, **"alma"** és **"körte"** szavak és az üres sztring (**""**) így tárolódnak el **str** –ben:



Sztringek C-ben

- Látható tehát, hogy a karaktertömb számára lefoglalt hely valójában csak egy felső korlátot jelent a sztring hosszára nézve. A sztring aktuális értéke ettől a korláttól lefelé bármikor eltérhet.
- A sztringek hosszának meghatározása viszont gyakran szükséges. Ezt a következő módokon tehetjük meg.

Sztringek

- A karaktersorozat hosszának meghatározására egy függvényműveletet írunk:

```
int strlen(char s[])
            /* s hosszának kiszámítása */
{
    int i = 0;
    while (s[i] != '\0') {
        ++i;
    }
    return i;
}
```

Sztringek

- Vagy a C nyelv automatikus konverzióit kihasználva:

```
int strlen(char s[])
                /* s hosszának kiszámítása */
{
    int i = 0;
    while (s[i] != 0) {
        ++i;
    }
    return i;
}
```

Sztringek

- Még több automatikus konverziót kihasználva:

```
int strlen(char s[])
                /* s hosszának kiszámítása */
{
    int i = 0;
    while (s[i]) {
        ++i;
    }
    return i;
}
```

Sztringek megadása

- A sztringek valóban karaktertömbök, tehát megadhatók úgy, mint egy tömb:

```
{'s','z','t','r','i','n','g','\0'}
```

- Van azonban egy egyszerűbb forma is:

```
"sztring"
```

- Az ilyen módokon megadott sztringkonstansok használhatók inicializálásra, azaz:

```
char str[20] = "sztring";
```

vagy akár

```
char str[] = "sztring";
```

Sztringek megadása

- Hogyan adjunk meg olyan sztringet, amelyben szerepel az idézőjel?

```
{'_', '"', '_', '\'', '_', '\0'}
```

- Mint a karaktereknél, itt is használható az escape karakter:

```
"_\"_'_"
```

- Az összes escape szekvencia használható, ami a karaktereknél. Az egyetlen különbség, hogy a kétfajta határoló karakter közül csak „sajátot” kell escape szekvenciával megadni.

Műveletek sztringekkel

- A sztringeken értelmezett alapvetőbb műveletek:
  - Hossz
    - A sztring hosszának meghatározása
  - Értékadás
    - A sztring értékének egyenlővé tétele egy másik sztringgel
  - Összefűzés
    - Két sztring összefűzése
  - Összehasonlítás
    - Két sztring lexikografikus összeahsonlítása
  - Karakter elérése
    - A sztring egy betűjének közvetlen elérése

Műveletek sztringekkel

- Mivel a sztring nem elemi típus, hanem egy speciálisan értelmezett tömb, a műveleteit függvényekkel valósították meg.
- Ezeket a függvényeket az

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

sor megadása után tudjuk használni.

Műveletek sztringekkel

- Hossz

  `size_t strlen(const char *s);`

  – Visszaadja az **`s`** hosszát.
- Értékadás

  `strcpy(char *dest, const char *src);`

  – Átmásolja az **`src`** értékét a **`dest`**-be. A sima értékadás művelet (**`=`**) tömbökről lévén szó nem működik. Az a programozó felelőssége, hogy a **`dest`** tömb elég hosszú legyen ahhoz, hogy az **`src`** értékét tárolni tudja.

Műveletek sztringekkel

- Összefűzés

  `strcat(char *dest, const char *src);`

  – Hozzáfűzi az **`src`** értékét a **`dest`**-hez. Az a programozó felelőssége, hogy a **`dest`** tömb elég hosszú legyen ahhoz, hogy az összefűzött értéket tárolni tudja.

- Összehasonlítás

  `int strcmp(const char *s1,`
  `const char *s2);`

  – Lexikografikusan összehasonlítja a két sztringet, és -1, 0 vagy 1 értékkel tér vissza attól függően, hogy **`s1`**$<_{lex}$**`s2`**, **`s1`**$=_{lex}$**`s2`** vagy **`s1`**$>_{lex}$**`s2`**.

Műveletek sztringekkel

- Karakter elérése
  - Ez a tömböknél szokásos **`[]`** operátorral valósítható meg.

- Egyéb műveletek is meg vannak valósítva a **`string.h`** –ban, ezeket is lehet használni.
- Linux alatt meg lehet nézni ezeket a függvényeket:

```
man string
```

POINTEREK ÉS TÖMBÖK

Pointerek és tömbök

- A programok megértéséhez beszélnünk kell a pointerek és a tömbök kapcsolatáról.
- A C nyelvben szoros kapcsolat van a mutatók és a tömbök között: valamennyi művelet, amely tömbindexeléssel végrehajtható, mutatók használatával éppúgy elvégezhető.
- Általában az utóbbi változat gyorsabb, de különösen a kezdők számára első ránézésre nehezebben érthető.

Pointerek és tömbök

- Az

  ```
  int a[10];
  ```

  deklaráció definiál egy **`10*sizeof(int)`** bájtos memóriaterületet, melynek egyes elemeire hivatkozhatunk az

  ```
  a[0], a[1], ... , a[9]
  ```

  változóhivatkozásokkal.

Pointerek és tömbök

- Ha **`pa`** deklarációja

  ```
  int *pa;
  ```

  akkor a

  ```
  pa = &a[0];
  ```

  értékadás úgy állítja be **`pa`**-t, hogy az az **`a`** nulladik elemére mutasson, vagyis **`pa`** az **`a[0]`** elem címét tartalmazza.
- Ekkor az

  ```
  x = *pa;
  ```

  értékadás **`a[0]`** tartalmát **`x`**-be másolja.

Pointerek és tömbök

- Ha **`pa`** egy tömb adott elemére mutat, akkor definíció szerint

  `pa+1`

  a tömb következő elemére mutat.
- Általában

  `pa-i` a pa előtti i. elemre,

  `pa+i` a pa mögötti i. elemre

  mutat.

Pointerek és tömbök

- Így ha **`pa`** az **`a[0]`**-ra mutat, akkor
  - **`*(pa + 1)`**
    `a[1]` tartalmát szolgáltatja,
  - **`pa+i`**
    az **`a[i]`** elem címe
  - **`*(pa+i)`**
    az **`a[i]`** elem értéke
- Ezek a megállapítások a tömbben elhelyezkedő változók típusától vagy méretétől függetlenül mindig igazak.

Pointerek és tömbök

- A pointeraritmetika alapdefiníciója, hogy a növekmény mértékegysége annak az objektumnak a tárbeli mérete, amire a mutató mutat.
- Az indexelés és a pointeraritmetika között láthatóan nagyon szoros kapcsolat van.
- Gyakorlatilag a tömbre való hivatkozást a fordító a tömb kezdetét megcímző mutatóvá alakítja át.

Pointerek és tömbök

- Ennek hatására a tömb neve nem más, mint egy mutatókifejezés, amiből számos hasznos dolog következik.
- Mivel a tömb neve ugyanaz, mint az illető tömb nulladik elemének címe, a

```
pa = &a[0];
```

értékadás úgy is írható, mint

```
pa = a;
```

Pointerek és tömbök

- Ez azt jelenti, hogy az

  `a[i]`

  hivatkozás írható

  `*(a+i)`

  alakban is: **`a[i]`** kiértékelésekor a C fordító azonnal átalakítja ezt **`*(a+i)`**-vé; a két alak teljesen egyenértékű.

Pointerek és tömbök

- Ha mindkét elemre alkalmazzuk az **`&`** operátort, akkor azonnal következik, hogy

  `&a[i]`

  és

  `a+i`

  szintén azonosak: **`a+i`** az **`a`**-t követő **`i`**-edik elem címe.

Pointerek és tömbök

- Másrészről, ha

  `pa`

  mutató, és

  `pa = a;`

  akkor azt kifejezésekben indexelhetjük:

  `pa[i]`

  ugyanaz mint

  `*(pa+i)`

Pointerek és tömbök

- Röviden: bármilyen tömb vagy indexkifejezés leírható, mint egy mutató plusz egy eltolás és viszont.
- Van azonban egy fontos különbség a tömbnév és a mutató között:
  - A mutató változóhivatkozás, így **`pa=a`**, **`pa++`** értelmes műveletek.
  - A tömbnév azonban nem változóhivatkozás, így **`a=pa`**, **`a++`** vagy **`p=&a`** nem megengedettek!

Pointerek és tömbök

- Amikor a tömbnév egy függvénynek adódik át, a függvény valójában a tömb kezdetének címét kapja meg.
- A hívott függvényen belül a formális paraméter tehát egy címet tartalmazó változó.
- Így már érthető a sztringkezelő függvények definíciója.

Pointerek és tömbök

- A függvénydefinícióban

  ```
  char s[];
  ```

  és

  ```
  char *s;
  ```

  egyaránt szerepelhet formális paraméterként; azt, hogy melyiket használjuk, nagymértékben az dönti el, hogy miként írjuk le a kifejezéseket a függvényen belül.

Pointerek és tömbök

- Amikor a tömbnév adódik át valamelyik függvénynek, a függvény tetszése szerint hiheti azt, hogy tömböt vagy mutatót kapott, és ennek megfelelően kezelheti azt.
- Akár mindkét típusú műveletet használhatja, ha ez célszerűnek és világosnak látszik.
- Lehetőség van arra, hogy a tömbnek csupán egy részét adjuk át valamelyik függvénynek oly módon, hogy a résztömb kezdetét megcímző mutatót adunk át.

Pointerek és tömbök

- Ha pl. **`a`** egy tömb neve, akkor

  `f(&a[2])`

  és

  `f(a+2)`

  is az **`a[2]`** elem címét adja át a függvénynek, mivel **`&a[2]`** és **`a+2`** egyaránt mutatókifejezés, mindkettő az **`a`** tömb harmadik elemére mutat.

Pointerek és tömbök

- A függvény deklarációjában akár

  ```
  f(int arg[]) { ... }
  ```

  akár

  ```
  f(int *arg) { ... }
  ```

  is lehet.
- Ami **`f`**-et illeti, az a tény, hogy az argumentum valójában egy nagyobb tömb egy részére vonatkozik, semmiféle következménnyel sem jár.

Pointerek és tömbök

- Hogyan adhatunk meg kétdimenziós tömböt paraméterként?
- A sorok száma nem érdekes, de az oszlopok számát meg kell adni a helyes címszámítás érdekében.
- Tekintsük az **`a[5][35]`** tömböt.
- Lehet így megadni a formális paraméter típusát

```
f(int a[][35]) { ... }
```

vagy így

```
f(int (*a)[35]) { ... }
```

Pointerek és tömbök

- A zárójelezés szükséges, mert `[]` magasabb prioritású művelet, mint a `*` művelet.
- Vagyis
  - `int (*a)[35];`

    egy a pointert deklarál, ami egy tömbre mutat amelyik 35 egészből áll.
  - `int *a[35];`

    egy a tömböt deklarál, ami 35 pointerből áll és a pointerek egészekre mutatnak.

Pointerek és tömbök

- A következő példa talán rávilágít tömbök és mutatók között meglévő kis különbségre.
- Tekintsünk a következő deklarációt!

```
char *honap[12];
char Honap[12][20];
```

- Szabályos változóhivatkozások a

  `honap[3][4]`

  is és

  `Honap[3][4]`

  is.

Pointerek és tömbök

- A **`Honap`** összesen 240 karakterből áll.
- A **`honap`** 12 pointerből áll, amelyek mutathatnak 20 hosszú sztringekre, ekkor neki is 240 karaktere van, plusz a 12 mutató. De lehet kevesebb karaktere is.

Pointerek és tömbök

- Deklarálhatunk így is azonnal kezdőértéket adva a változóknak:

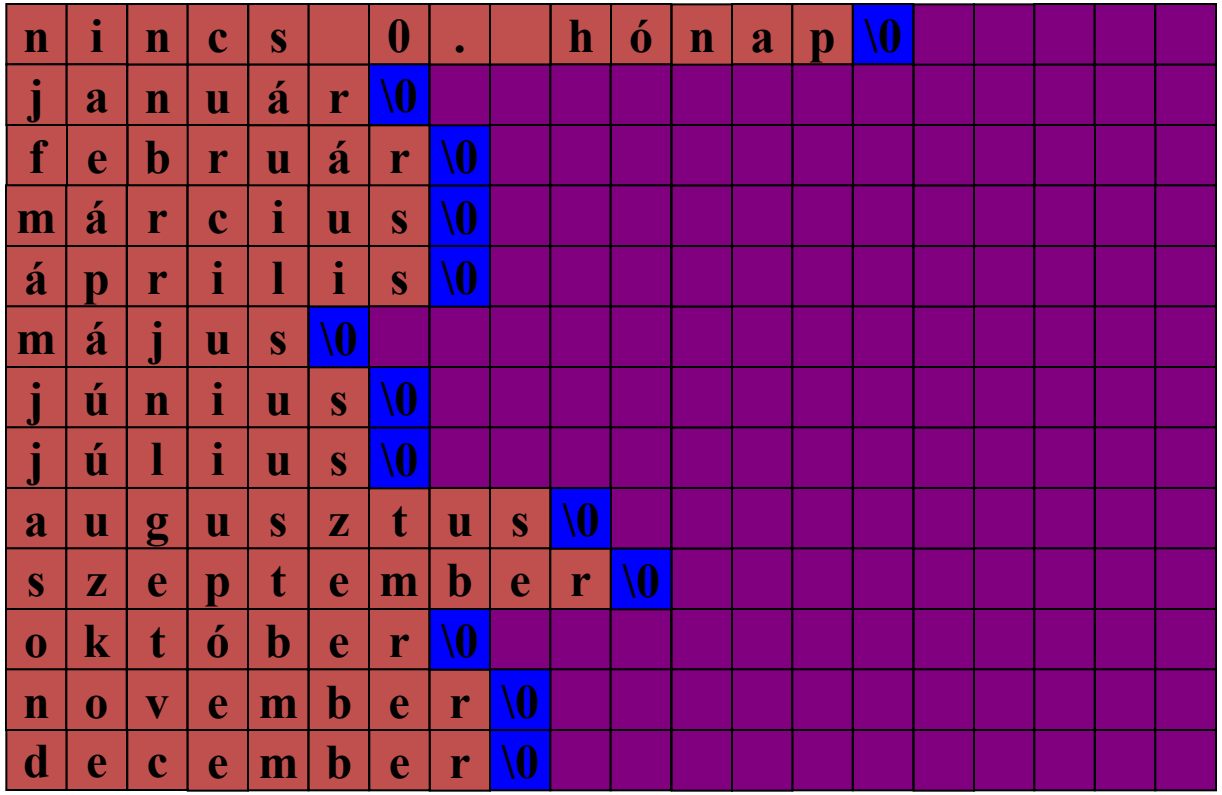
```
char *honap[] = { "nincs 0. hónap",
     "január", "február", ... , "december" };

char Honap[][20] = { "nincs 0. hónap",
     "január", "február", ... , "december" };
```

- Látható, hogy a **`honap`** pointerei különböző hosszúságú szövegekre mutathattak.

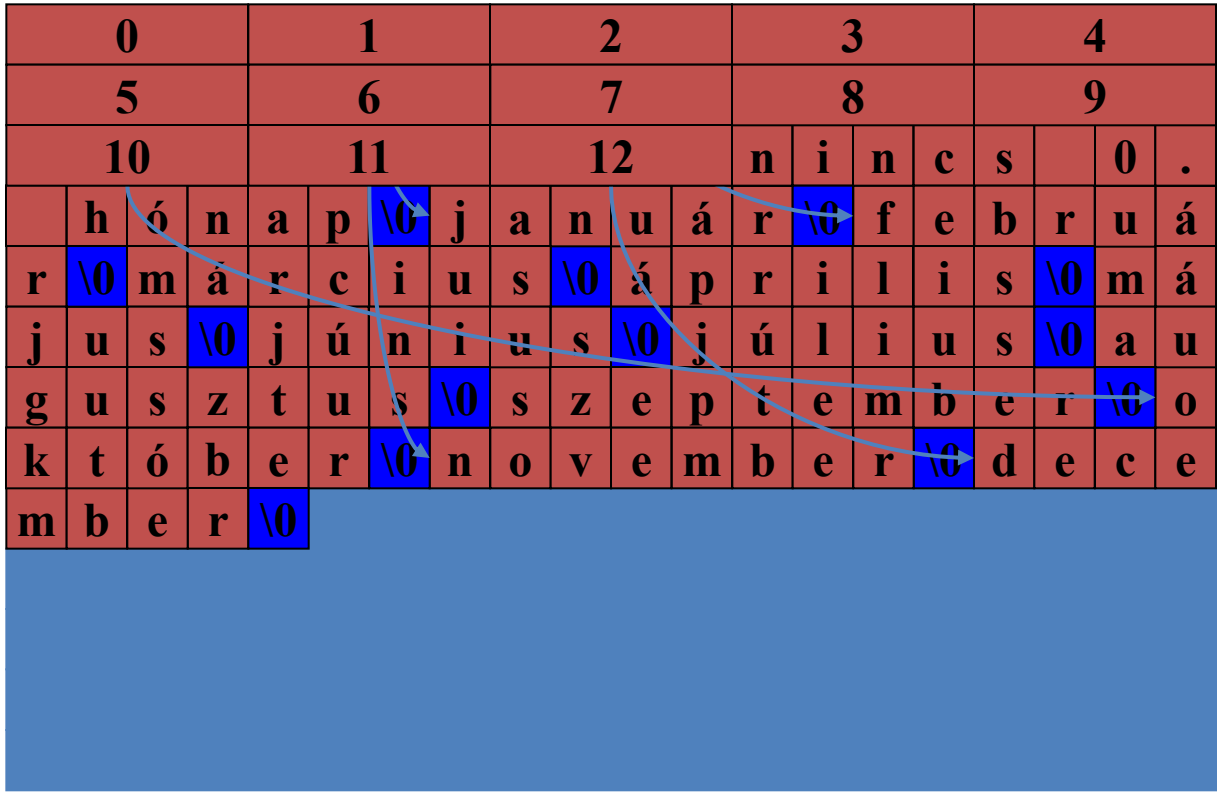
Pointerek és tömbök

- **`char Honap[][20]`**



Pointerek és tömbök

- `char *honap[]`



Pointerek és tömbök

- Ha tömböt kezdőértékkel szeretnénk deklarálni, akkor elhagyható az első dimenzió megadása, mert a kezdőértékek számából ez adódik.
- Ha kevesebb kezdőértéket adunk meg, mint amekkora tömbre később szükségünk van, akkor természetesen minden méretet meg kell adni.

Pointerek és tömbök

```
int t[][3] = {
    { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9 }
}; /* A t tömb 3x3-as lesz. */
```

```
int t[4][3] = {
    { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9 }
}; /* A t tömb 4x3-as lesz és
      az utolsó sor minden eleme 0 lesz. */
```

```
int t[][3] = {
    { 1 },
    { 4 },
    { 7 },
    { 9 }
}; /* A t tömb 4x3-as lesz,
      az első oszlopot adtuk meg, a többi elem 0 lesz. */
```

Pointeraritmetika

- Ha **`p`** mutató, akkor

  `p++ (p--)`

  oly módon inkrementálja **`p`**-t, hogy az a megcímzett tetszőleges típusú objektum következő (előző) elemére,

  `p += i (p -= i)`

  pedig úgy, hogy az a pillanatnyilag megcímzett elem utáni (előtti) **`i`**-edik elemre mutasson.
- Alkalmazhatjuk tehát a pointerekre az egész hozzáadás és kivonás műveleteket.

Pointeraritmetika

- Mutatók kivonása szintén megengedett: ha **`p`** és **`q`** ugyanannak a tömbnek az elemeire mutatnak, akkor

  `p-q`

  a **`p`** és **`q`** közötti elemek darabszáma, ami csak akkor egyezik meg a két memóriacím tényleges különbségével, ha az elemek egy bájtosak.

Pointeraritmetika

- E tényt kihasználva megírhatjuk a strlen újabb változatát:

```
int strlen (char *s)
/* kiszámítja az s karakterlánc hosszát */
{
    char *p = s;
    while (*p != '\0') {
        p++;
    }
    return (p - s);
}
```

Pointeraritmetika

- A deklarációban **`p`** kezdeti értékeként **`s`**-et adtuk meg, vagyis **`p`** kezdetben az **`s`** első karakterére mutat.
- A **`while`** ciklusban addig vizsgáljuk az egymást követő karaktereket, amíg a véget jelző `'\0'` elő nem kerül.

Pointeraritmetika

- Mivel `'\0'` értéke nulla, és ez a hamis értéknek felel meg, elhagyható az explicit vizsgálat. Az ilyen ciklusokat gyakran az alábbi alakokban írják:

```
while (*p) { p++; }
while (*p) p++;
```

- Minthogy **p** karakterekre mutat, **p++** minden alkalommal a következő karakterre lépteti **p**-t, és **p - s** az átlépett karakterek számát, vagyis a karakterlánc hosszát adja meg.

Pointeraritmetika

- Az említett műveleteken kívül (mutató és integer összeadása és kivonása, két mutató kivonása és összehasonlítása) minden más mutatóművelet tilos!
- Nincs megengedve két mutató összeadása, szorzása, osztása, mutatók léptetése, maszkolása, sem pedig **`float`** vagy **`double`** mennyiségeknek mutatókhoz történő hozzáadása.

Pointeraritmetika

- Említést kell tenni a típustalan pointerről is:

  ```
  void *p;
  ```

- Ekkor a **`p++`** egy bájttal való növelést jelent.

- A NULL konstans így lehet deklarálva a **`stdio.h`**-ban:

```
#define NULL 0
```

vagy

```
#define NULL ((void*)0)
```

REKORD TÍPUS

Rekord típus

- A tömb típus nagyszámú, de ugyanazon típusú adat tárolására alkalmas.
- Problémák megoldása során viszont gyakran előfordul, hogy különböző típusú, de logikailag összetartozó adatelemek együttesével kell dolgozni.
- Az ilyen adatok tárolására szolgálnak a rekord típusok, ezek létrehozására pedig a rekord típusképzések

Szorzat-rekord típus

- Ha az egyes típusú adatokat egyszerre kell tudnunk tárolni, szorzat-rekordról beszélünk.

- Legyenek T_1, ... ,T_k tetszőleges típusok.
- A T_1, ... ,T_k típusokból képezzük a

  $T= T_1 x \dots x T_k = \{(a_1, \dots, a_k) \mid a_1 \text{ in } T_1, \dots, a_k \text{ in } T_k\}$

  értékhalmazt, tehát a T_1, ... ,T_k típusok direktszorzatát.

Szorzat-rekord típus

- A T halmazon is értelmezhetünk kiolvasó és módosító műveletet, mint a tömb típus esetén, de most nem adhatunk meg index értéket, mert a különböző sorszámú elemek itt eltérő típusúak lehetnek.
- Ehelyett bevezetünk **k** számú kiolvasó és módosító műveletet.
- Az új adattípusra a T=Rekord(T_1, ... ,T_k) jelölést használjuk és szorzat-rekordnak vagy csak egyszerűen rekordnak nevezzük.

Szorzat-rekord műveletei

- Kiolvas$_i$(-> A:T; <- X:T$_i$), (i in 1..k)
  - A művelet végrehajtása után X=a_i, ha A=(a_1, ... ,a_k).
- Módosít$_i$(<-> A:T; -> X:T$_i$), (i in 1..k)
  - Ha a művelet végrehajtása előtt
    A=(a_1, ... ,a_{i-1},a_i,a_{i+1}, ... ,a_k),
    akkor a művelet végrehajtása után
    A=(a_1, ... ,a_{i-1},x,a_{i+1}, ... ,a_k).
- Az X=Y értékadás értelmezett, ha az X és Y változók típusa T.

Szorzat-rekord C nyelven

- A T=Rekord(T_1, ... ,T_k) típust C-ben a **`struct`** kulcsszóval definiáljuk:

```
typedef struct T {
  T1 M1;
  ...
  Tk Mk;
} T;
```

- A fenti típusképzésben az **`M1`**,…,**`Mk`** azonosítókat mezőazonosítóknak (tagnak, member-nek) hívjuk és lokálisak a típusképzésre nézve.

Szorzat-rekord C nyelven

- Az absztrakt típus műveletei mezőhivatkozások segítségével valósíthatóak meg, aminek műveleti jele a

  `.`

- Minden **`T`** típusú **`A`** változó esetén léteznek az

  `A.M1, ... ,A.Mk`

  mezőhivatkozások úgy, hogy A=(A.M_1, ... ,A.M_k), továbbá A.M_i (i in 1..k) közönséges T_i típusú változónak tekintődik.
- A `.` balasszociatív és a legmagasabb prioritású.

Szorzat-rekord C nyelven

- A prioritási előírás csökkenő sorrendben
  - elemkiválasztások ([], .)
  - a egyoperandusú műveletek (-, ++, --, !, ~, &, *)
  - a multiplikatív műveletek (*, /, %)
  - az additív műveletek (+, -)
  - bitléptetés (<<, >>)
  - a kisebb-nagyobb relációs műveletek (<=, >=, <, >)
  - az egyenlő-nem egyenlő relációs műveletek (==, !=)
  - bitenkénti 'és' művelet (&)
  - bitenkénti 'kizáró vagy' művelet (^)
  - bitenkénti 'vagy' művelet (|)
  - a logikai 'és' művelet (&&)
  - a logikai 'vagy' művelet (||)
  - a feltételes művelet (? :)
  - értékadó művelet (=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)
  - szekvencia művelet (,)

Szorzat-rekord C nyelven

- Kiolvas$_i$(A,X)

  `X=A.Mi`

- Módosít$_i$(A,X)

  `A.Mi=X`

- A mezőlista szintaxisa lehetővé teszi, hogy az azonos típusú mezőket összevonva deklaráljuk úgy, hogy vesszővel elválasztva felsoroljuk a mezőazonosítókat (mint a változódeklarációnál).

Szorzat-rekord példák

```
typedef struct DatumTip {
    short ev;
    char ho;
    char nap;
} DatumTip;

typedef char Szoveg20[21];

typedef struct CimTip {
    Szoveg20 varos, utca;
    short hazszam;
    short iranyitoSz;
} CimTip;
```

```
typedef struct SzemelyTip {
    struct {
        Szoveg20 csaladi, uto;
    } nev;
    int szemelyiSzam;
    Szoveg20 szulHely;
    Datumtip szulIdo;
    CimTip lakcim;
} SzemelyTip;
```

Szorzat-rekord példák

- Deklaráljuk az **`x`**-t **`SzemelyTip`** típusú változónak!

```
SzemelyTip x;
```

- Ekkor az **`x`** változóban tárolt adatok közül például a személy vezetéknevére az

```
x.nev.csaladi
```

mezőhivatkozással hivatkozhatunk, születési évét pedig az

```
x.szulIdo.ev
```

mező tartalmazza.

Szorzat-rekord gépi megvalósítása

- A **`struct`** típusú változó számára foglalt memória mérete, amely a **`sizeof`** függvénnyel lekérdezhető:

```
sizeof(E) =
    sizeof(T1) + ... + sizeof(Tk) +
        igazítás
```

- Valamennyi változati mező egymást (az esetleges igazítást is figyelembe véve) követő, növekvő memóriacímen kezdődik.

Egyesített-rekord

- Ha az egyes típusú adatokat nem kell egyszerre tárolni, egyesített-rekordról beszélünk.

- Legyenek T_1, ... ,T_m tetszőleges típusok.
- A T_1, ... ,T_m típusokból képezzük az

  $E= T_1+ \ldots +T_m =$

  $\{1\}xT_1 \cup \ldots \cup \{m\}xT_m = \{(i,a) \mid i \text{ in } 1..m, a \text{ in } T_i\}$

  értékhalmazt, tehát a T_1, ... ,T_m típusok értékhalmazainak diszjunkt egyesítését.

Egyesített-rekord

- E elemei tehát olyan rendezett párok, amelyeknek első komponense meghatározza, hogy a második komponens melyik típusból való érték.
- Általában a következő konstrukciót alkalmazhatjuk.
- Legyen T_0 olyan típus, amely tartalmazza a $c_1,...,c_m$ különböző értékeket, továbbá legyenek $T_1,...,T_m$ tetszőleges típusok.

Egyesített-rekord

- Képezzük az

$$E = \{c_1\}xT_1 \cup \ldots \cup \{c_m\}xT_m = \{(c_i,a) \mid c_i \text{ in } T_0, a \text{ in } T_i, i \text{ in } 1..m\}$$

  halmazt.
- A E értékhalmazt a következőkben definiált műveletekkel a T_0 változati típusból és a T_1, ... ,T_m egyesítési-tag típusból képzett egyesített rekord típusnak nevezzük.

Egyesített-rekord műveletek

- Változat(-> A:E; <- v:T_0)
  - A művelet végrehajtása után v=c_i, ha A=(c_i,a).
- Kiolvas$_i$(-> A:E; <- X:T_i), (i in 1..m)
  - A művelet végrehajtása után X=a, ha A=(c_i,a).
  - A művelet hatástalan, ha A első komponense nem c_i.
- Módosít$_i$(<-> A:E; -> X:T_i) , (i in 1..m)
  - A művelet végrehajtása után A=(c_i,x).

Egyesített-rekord C nyelven

- Az egyesített-rekord típust C-ben az **`union`** típusképzéssel valósítjuk meg.
- Mivel az **`union`** konstrukcióban nincs lehetőség a jelzőmező megadására, ezért ha szükségünk van a jelzőmezőre is, akkor a C-ben az előző konstrukcióhoz folyamodunk.
- C-ben jelzőmezőnek alkalmas az **`int`**, vagy a **`char`** típus valamelyik változata, vagy az általunk definiált **`enum`** típus.

Egyesített-rekord C nyelven

- Az **E** egyesített-rekord típust a következőképpen kell definiálni:

```
typedef ... T0;
typedef struct E {
    T0 Milyen;
    union {
        T1 V1;
           ...
        Tm Vm;
    };
} E;
```

Egyesített-rekord C nyelven

ahol

- **`Milyen`** azonosító a változati (jelző) mezőazonosító
- **`T0`** a változati (jelző) típus azonosítója
- **`V1, ... ,Vm`** az egyesítési tag mezőazonosítók
- **`T1, ... ,Tm`** az egyesítési tag típusok

Egyesített-rekord C nyelven

- A műveletek megvalósítása itt is mezőhivatkozással történik.
- Változat(A,v)

```
v = A.Milyen
```

- Kiolvas$_i$(A,x)

```
x = A.Vi
```

- Módosít$_i$(A,x)

```
{ A.Milyen=ci; A.Vi=x }
```

Egyesített-rekord C nyelven

- Az egyesített-rekord C-ben történő megvalósításában mindig hivatkozhatunk az **`A.V`**$_{\mathbf{i}}$ egyesítési tag mezőre, függetlenül attól, hogy az **`A.milyen`** változat mező aktuális értéke **`c`**$_{\mathbf{i}}$ vagy sem.
- Ha a típusképzésben nem adunk meg változati mezőazonosítót, akkor nincs lehetőségünk az aktuális változatról információ tárolására és lekérdezésére; ekkor a **`struct`** konstrukció el is maradhat és csupán az **`union`** rész marad.

Egyesített-rekord példák

```
typedef enum {
    kor, haromszog, negyszog
} Sikidom;

typedef union Szam {
    double Valos;
    long int Egesz;
} Szam;
```

Egyesített-rekord példák

```
typedef struct Idom {
  Sikidom Fajta;
  union {
    double Sugar;
    struct {
      double A, B, C;
    } U1;
    struct {
      double D1, D2, D3, D4;
    } U2;
  } UU;
} Idom;
```

```
typedef struct Alakzat {
  double x, y; /* a koordináták */
  Sikidom forma;
  union {
    double sugar;
    struct {
      double alfa, oldal1, oldal2;
    };
    struct {
      double hossz, szel;
    };
  };
} Alakzat;
```

Egyesített-rek. gépi megvalósítása

- Az **`union`** típusú változó számára foglalt memória mérete, amely a **`sizeof`** függvénnyel lekérdezhető:

  `sizeof(E) =`
  `Max{sizeof(T1),…,sizeof(Tm)}`

- Ha az egyesített-rekord változati mezőt is tartalmaz, ennek méretét a **`struct`** gépi megvalósítása szerint hozzá kell adni ehhez.
- Valamennyi változati mező ugyanazon memóriacímen kezdődik.

struct, union

- Mint az a példákból látható volt, a **`struct`** és az **`union`** deklarációk egymásba ágyazhatóak.
- Ilyen esetekben a rekord típusú mező mezőazonosítója elhagyható, ekkor ezen mező mezőazonosítói úgy látszanak, mintha a külső rekord típus mezőazonosítói lennének:

```
Idom I;     /* Vannak mezőazonosítók a   */
I.UU.U1.B; /* rekord mezőkhöz: UU,U1,U2 */
Alakzat A; /* Nincsenek mezőazonosítók  */
A.alfa;     /* a rekord mezőkhöz */
```

struct, union

- A **`struct`** vagy az **`union`** kulcsszót struktúracímke (struktúranév) követheti.
- Ez egy név, amely megnevezi az adott típusú struktúrát, vagy uniót és a továbbiakban rövidítésként használható a **`{ }`**-ben lévő részletes deklaráció helyett.
- A struktúranév és a típusnév meg is egyezhet, mint a fenti példákban, így a legrugalmasabb a felhasználásuk. Pl.:

```
struct Alakzat a1, a2;
Alakzat b1, b2;
```

struct, union

- A **`struct`** és az **`union`** típusú változók esetén megengedett az értékadás művelet, így függvényargumentumként használhatjuk ezeket (érték fajtájú paraméterkezelés) illetve a függvényművelet eredményének típusa is lehet **`struct`** vagy **`union`**.
- Megengedett az **`&`** művelet használata is, így a **`struct`** és az **`union`** típusú változók változó fajtájú paraméterként is kezelhetők.

struct, union

- A **`struct`** és **`union`** típusú változók is kaphatnak kezdőértéket.
- Ha minden tagot inicializálunk, akkor a **`{ }`** elmaradhatnak, különben pedig ugyanott vannak, ahol a stuktúra deklarációjában.
- Az **`union`** típusnak csak az első tagja inicializálható.

Bitmezők

- A C nyelv lehetővé teszi egy byte-on belüli bitek elérését magas szinten, bitmezős struktúrák segítségével.
- A megvalósítás nagyon gépfüggő.

- Felhasználhatjuk pl. hardver-programozáshoz szükséges bitsorozatok magas szintű kezelése (driverek írásához), vagy jelzőbitek (flag-ek) tömör elhelyezésére.

Bitmezők

- A bitmező a struktúrához hasonlít, csak az egyes tagok után a szükséges bitek száma is meg van adva. Pl.

```
struct {
    unsigned int flag1 : 1;
    unsigned int flag2 : 1;
    unsigned int flag3 : 2;
} jelzok;

jelzok.flag1 = jelzok.flag2 = 0;
jelzok.flag3 = 1;

if (jelzok.flag1 == 0 && jelzok.flag3 == 1) {
    /* ... */
}
```

Bitmezők

- Miért jó ez?

```
struct {
    unsigned short int flag1 : 1;
    unsigned short int flag2 : 1;
    unsigned short int flag3 : 2;
    unsigned short int flag4 : 4;
    unsigned short int flag5 : 2;
    unsigned short int flag6 : 6;
} jelzok;
```

- Bitmezők nélkül ez legalább 6 bájt lenne, így viszont csak 2.

Struct, Union, Enum C nyelven

- A **`struct`**, **`union`** és **`enum`** a típusdeklaráció, típusdefiníció szempontjából hasonlóan működik, így amit a következőkben a **`struct`** típusról elmondunk, az analóg módon a **`union`** és **`enum`** típusokra is érvényes.

Struct, Union, Enum

- Változódeklarációk:

```
struct {
    int a, b;
} vstruct;

union {
    long long l;
    double d;
} vunion;

enum {
    nulla, egy, ketto, harom
} venum;
```

Struct, Union, Enum

- Ha ugyanilyen típusú változókat szeretnénk később is deklarálni akkor érdemes elnevezni a struktúrát:

```
struct s {
    int a, b;
} vstruct;
...
struct s masodik;
...
struct s harmadik;
```

Struct, Union, Enum

- De megtehetjük azt is, hogy egyszerűen csak definiáljuk a struktúrát (változódeklaráció nélkül), és később használjuk fel:

```
struct s {
    int a, b;
};
...
struct s masodik;
...
struct s harmadik;
```

Struct, Union, Enum

- Sőt, megtehetjük azt is, hogy egyelőre csak deklaráljuk a struktúrát, és csak később definiáljuk (de a definíció NEM maradhat el!):

```
struct s;

...

struct s masodik;
...
struct s harmadik;
...
struct s {
    int a, b;
} vstruct;
```

Struct, Union, Enum

- Változódeklaráció helyett készíthetünk típusdefiníciót is:

```
typedef struct {
    int a, b;
} S;

...
S masodik;
S harmadik;
```

Struct, Union, Enum

- Akár így is:

```
struct s {
    int a, b;
} vstruct;
typedef struct s S;
...
struct s masodik;
S harmadik;

s negyedik;       /* ROSSZ !!! */
struct S otodik; /* ROSSZ !!! */
```

Struct, Union, Enum

- Vagy így:

```
typedef struct s {
    int a, b;
} S;

...
struct s masodik;
S harmadik;

s negyedik;       /* ROSSZ !!! */
struct S otodik; /* ROSSZ !!! */
```

Struct, Union, Enum

- De adhatjuk a struktúrának és a típusnak ugyanazt a nevet is:

```
typedef struct st {
    int a, b;
} st;

...
struct st masodik;
st harmadik;
```

-> operátor

- Struktúrára mutató pointer esetén szükségünk lesz olyan hivatkozásokra, mint

```
(*tp).nev
(*tp).fuggveny
```

  ahol **`tp`** egy struktúrára mutató pointer.
- A zárójelezésre szükség van, mivel a mezőkiválasztás (**`.`**) magasabb precedenciájú, mint a dereferencia (**`*`**).

-> operátor

- Mivel a C nyelvben sokszor van szükség az ilyen jellegű hivatkozásokra, ezért egy új műveletet vezetünk be.
- Ennek műveleti jele

  `->`

  és egy pointer által megmutatott struktúra egy mezőjének kiválasztására alkalmas.
- A prioritási sorban legfelül, a `.` művelet mellett helyezkedik el.

-> operátor

- Az operátor segítségével tehát a

  ```
  (*tp).nev
  (*tp).fuggveny
  ```

  alakú hivatkozások egyszerűbben

  ```
  tp->nev
  tp->fuggveny
  ```

  alakban írhatóak.

TÍPUS KÉNYSZERÍTÉS

Típuskényszerítés

- A típuskényszerítés egy konverziós művelet, amelyet egy ()-be zárt típusmegadással írunk elő.
- A prioritási előírás csökkenő sorrendben
  - B elemkiválasztások és fgv. ([], ., ->, ())
  - J a egyoperandusú műveletek (-, ++, --, !, ~, &, *, sizeof)
  - **J típuskényszerítés (())**
  - B a multiplikatív műveletek (*, /, %)
  - B az additív műveletek (+, -)
  - B bitléptetés (<<, >>)
  - B a kisebb-nagyobb relációs műveletek (<=, >=, <, >)
  - B az egyenlő-nem egyenlő relációs műveletek (==, !=)
  - B bitenkénti 'és' művelet (&)
  - B bitenkénti 'kizáró vagy' művelet (^)
  - B bitenkénti 'vagy' művelet (|)
  - B a logikai 'és' művelet (&&)
  - B a logikai 'vagy' művelet (||)
  - J a feltételes művelet (? :)
  - J értékadó művelet (=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)
  - B szekvencia művelet (,)

Típuskényszerítés

- Példa típuskényszerítésre:
  - Két **`int`** tipusú változó osztása egészosztás és **`int`** eredményt ad.

```
int i = 7; j = 2;
double x;
x = i / j;                  /* x értéke 3.0 lesz */
```

Ha nem egészosztást szeretnénk, akkor kényszerítsük az egyik operandust valamelyik valós típusúvá:

```
x = (double) i / j;  /* x értéke 3.5 lesz */
```

Típuskényszerítés

- Példa típuskényszerítésre:
  - A dinamikus változók számára leggyakrabban a **`malloc()`** függvénnyel foglalunk helyet. A függvény a **`malloc.h`**-ban van deklarálva:

```
typedef unsigned size_t;
void *malloc(size_t);
void free(void *);
void *realloc(void *, size_t);
void *calloc(size_t, size_t);
```

Típuskényszerítés

- A függvény lefoglalja a megadott bájtszámú területet és a címét visszadja. Ha nem sikerül a tárterület lefoglalása, akkor a visszaadott érték **`NULL`**.
- A **`malloc()`** függvény alkalmazásánál típuskényszerítést kell használni. Pl.

```
int *p;
p = (int *) malloc(100 * sizeof(int));
```

Típuskényszerítés

- Hasonlóan kell eljárni bonyolultabb típusok esetén:

```
typedef struct cellatip {
    Elemtip adat;              /* adatelem */
    struct cellatip *csat;
          /* a következő elem cellájára */
} cellatip;
typedef struct cellatip *pozicio;
                          /* pointer típus */
pozicio p;
p = (cellatip *) malloc(sizeof(cellatip));
```

Típuskényszerítés

- A típuskényszerítés egy konverziós művelet, tehát nem alkalmazható mindig. Pl.
  - Van egy **`void * p`** pointer. Tudom, hogy most **`int`**-re mutat. Szeretném a pointert léptetni. Lehet-e így?

    ```
    ((int *)p)++;
    ```

    Így nem lehet, de így már igen:

    ```
    p = (void *)((int *)p + 1);
    ```

    vagy egyszerűen így

    ```
    p += sizeof(int);
    ```

VÁLTOZÓ HIVATKOZÁS

Változóhivatkozás

- Az alapvetőbb típuskonstrukciók megismerése után visszatérhetünk a dinamikus változóknál megemlített három fogalomhoz:
  - változóhivatkozás
  - hivatkozott változó
  - változó értéke
- A C nyelvben a változóhivatkozás neve:

  `l-value`
- Értékadás bal oldalán vagy például a **`++`** operátor operandusaként csak ilyen **`l-value`** szerepelhet.

Változóhivatkozás

- A változóhivatkozás szintaktikus egység, tehát meghatározott formai szabályok szerint képzett jelsorozat egy adott programnyelven.
- A C nyelvben egy változóhivatkozás nagyon bonyolult is lehet, és alapvetően nem más, mint egy kifejezés.
- Azt, hogy mely kifejezések tekinthetők (szintaktikailag) érvényes változóhivatkozásnak, az alábbi szabályok alapján dönthető el.

Változóhivatkozás

- Minden olyan *változó azonosító*, amely az adott blokkban látható egyben **hivatkozás** is, és a típusa a változó deklarációjában megadott típus.
- Ha `L` egy `T` *nem tömb típusú hivatkozás*, akkor egyben `T` típusú **érvényes *változóhivatkozás*** is.
- Ha `L` egy `T` típusú *hivatkozás vagy* `T` típusú *érvényes változóhivatkozás*, akkor egyben `T` típusú **kifejezés** is.

Változóhivatkozás

- Ha **`X`** egy **`T`** típusú *érvényes változóhivatkozás*, és **`T struct`** vagy **`union`**, valamint **`m`** a **`T`** egyik **`E`** típusú mezője, akkor
  - **`T.m`**

  hivatkozás, típusa pedig **`E`**.
- Ha **`X`** egy **`T struct`** vagy **`union`** típusú *kifejezés*, valamint **`m`** a **`T`** egyik **`E`** típusú mezője, akkor
  - **`T.m`**

  `E` típusú **kifejezés**.

Változóhivatkozás

- Ha **`X`** egy **`T*`** vagy **`T[]`** típusú *kifejezés*, és **`i`** egy **`int`** típusú *kifejezés*, akkor
  - **`*X, X[i]`**

  érvényes **`T`** típusú **hivatkozások**, továbbá
  - **`X+i, X-i`**

  `T*` típusú **kifejezések**.
- Ha **`L`** egy **`T*`** típusú *érvényes változóhivatkozás*, akkor
  - **`++X, --X, X++, X--, X+=i, X-=i`**

  `T*` típusú **kifejezések**.

Változóhivatkozás

- Ha **`L`** egy **`T`** típusú *érvényes változóhivatkozás*, akkor
  - **`&L`**

  egy **`T*`** típusú **kifejezés**.
- Ha **`L`** egy **`T[n]`** típusú *hivatkozás*, *és* **`L`** *nem változó azonosító*, akkor
  - **`&L`**

  egy **`T(*)[n]`** típusú **kifejezés**, azaz egy **`T[n]`** típusra mutató pointer kifejezés.

Függvényre mutató pointer

- Eddig kétféleképpen közelítettük meg a pointer típusú változót:
  - Első közelítésben egy `p` pointer típusú változó értéke egy meghatározott típusú dinamikus változó.
  - A másik szerint a `p` pointer típusú változó értéke egy másik változóhoz tartozó memória mező címe.
- A harmadik megközelítésben a pointer egy részalgoritmusra is mutathat, amelyet aktuális paraméterekkel végrehajtva a megadott típusú eredményhez jutunk.

Függvényre mutató pointer

- Minden, a pointerekre megengedett művelet elvégezhető: szerepelhet értékadásban, elhelyezhető egy tömbben vagy rekordban, átadható egy függvénynek aktuális paraméterként, lehet egy függvény visszatérési értéke, stb.
- Ilyen típust egyszerűen úgy deklarálhatunk, hogy a megfelelő függvényfejben a függvény azonosítóját pointerre cseréljük, vagyis a típus azonosítója elé ***-t írunk.

Függvényre mutató pointer

- Mivel a ***** alacsonyabb priorítású, mint a **()**, ezért a dereferenciát zárójelpárba kell tenni.
- Legyen pl.:

```
double Sin2x(double x)
{
  return sin(2.0 * x);
}
```

egy ilyen függvényre mutató pointer típus:

```
typedef double (*FuggvenyTip)(double x);
```

vagy

```
typedef double (*FuggvenyTip)(double);
```

Függvényre mutató pointer

- Egy változódeklaráció kezdőérték megadásával

```
FuggvenyTip f = Sin2x;
```

- És a használata

```
double x, y;
y = (*f)(x);
```

Határozott integrál

- Problémafelvetés:
  - Adott függvény határozott integrálját közelítsük egy beolvasott intervallumon a felhasználó által megadott számú részre osztva az intervallumot.
- Specifikáció:
  - Input
    - Az A,B intervallum végpontjai
    - A részek száma
  - Output
    - Valós szám, a határozott integrál értéke

Határozott integrál

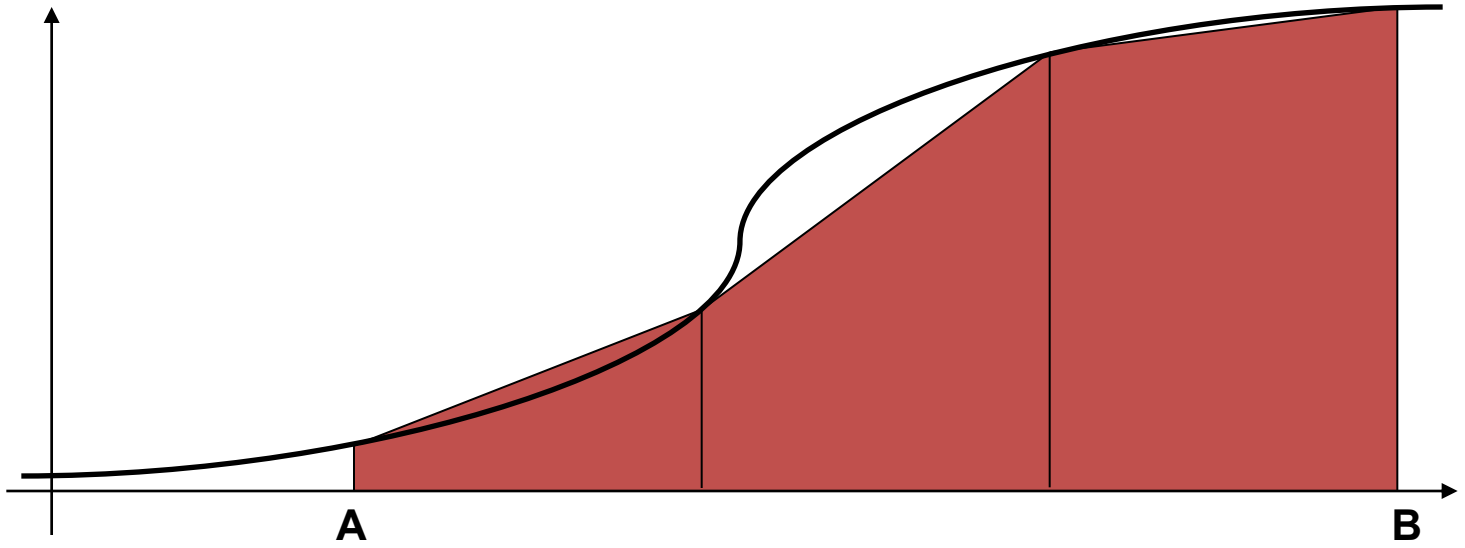
- Algoritmustervezés:
  - A fő algoritmusban csak az input adatokat kell beolvasni, az integrálandó függvényt az aktuális paraméterekkel meghívni, végül az eredményt kiiratni.

Integrált kiszámító függvény

- Problémafelvetés:
  - A paraméterként megadott függvény határozott integrálját számoljuk egy paraméterként adott intervallumon. Paraméterként adott az is, hogy az intervallumot hány részre kell osztani a közelítésnél.
- Specifikáció:
  - Input:
    - Az integrálandó függvény
    - Az a,b intervallum végpontjai
    - Az, hogy hány részre osszuk fel az intervallumot
  - Output:

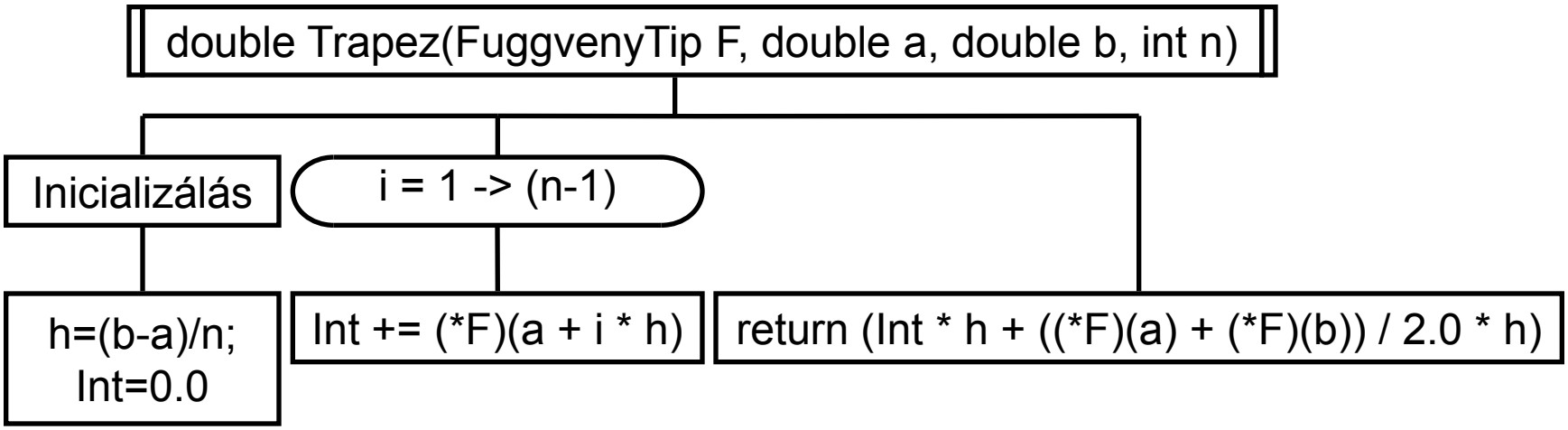
Integrált kiszámító függvény

- Algoritmustervezés
  - A trapéz módszer szerint történik a közelítés. A képlet egyszerű átalakításával egy számlált menetű ismétléses vezérlést kapunk.



Integrált kiszámító függvény

- Algoritmustervezés



Határozott integrál

```
/* Közelítő integrálás a trapéz szabály segítségével.
 * Az integrálandó függvényt paraméterként kapjuk.
 * 1998. Április 14.  Dévényi Károly, devenyi@inf.u-szeged.hu
 * 2006. Augusztus 14. Gergely Tamás, gertom@inf.u-szeged.hu
 */

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

                                /* Az integrálandó függvény típusa */
typedef double (*FuggvenyTip)(double x);
```

>>>

Határozott integrál

```
static double Trapez(FuggvenyTip F,   /* F(x)-t integráljuk az */
                     double a, double b,  /* a,b intervallumon */
                     int n)    /* n részre osztva az interv.-t */
{
         /* Közelítő integrálás a trapéz szabály segítségével. */
  double Int, h;
  int i;                                        /* a ciklusváltozó */
  h = (b - a) / n;
  Int = 0.0;
  for (i = 1; i < n; i++) {          /* fgv. értékek összegzése */
      Int += (*F)(a + i * h);
/*    Int += F(a + i * h);                    <- így is lehetne */
  }
  return (Int * h + ((*F)(a) + (*F)(b)) / 2.0 * h);
/*return (Int * h + (F(a) + F(b)) / 2.0 * h);<- így is lehetne */
}
```

>>>

Határozott integrál

```
static double Eadxperx(double x)
{                                  /* az első integrálandó függvény */
  return (exp(x) / x);
}

static double Sin2x(double x)
{                                 /* a második integrálandó függvény */
  return sin(2.0 * x);
}

main()
{
  double A, B;
  int n;
  printf("Az exp(x)/x függvény közelítő integrálja.\n");
  printf("Kérem az integrálási intervallumot ");
```

>>>

Határozott integrál

```
  printf("és az osztáspontok számát (A,B,n)!\n");
  printf("A:? "); scanf("%g%*[^\n]", &A); getchar();
  printf("B:? "); scanf("%g%*[^\n]", &B); getchar();
  printf("n:? "); scanf("%d%*[^\n]", &n); getchar();
  printf("A integrál közelítő értéke: ");
  printf("%10.5f\n", Trapez(Eadxperx, A, B, n));

  printf("Az sin(2x) függvény közelítő integrálja.\n");
  printf("Kérem az integrálási intervallumot ");
  printf("és az osztáspontok számát (A,B,n)!\n");
  printf("A:? "); scanf("%g%*[^\n]", &A); getchar();
  printf("B:? "); scanf("%g%*[^\n]", &B); getchar();
  printf("n:? "); scanf("%d%*[^\n]", &n); getchar();
  printf("A integrál közelítő értéke: ");
  printf("%10.5f\n", Trapez(Sin2x, A, B, n));
}
```

DINAMIKUS TÖMB

Dinamikus tömb típus

- Ha a programban deklarálunk egy tömböt, azzal az lehet a baj, hogy a méretét fordítási időben meg kell adni.
- Ez viszont nem mindig ismert, így előfordulhat, hogy a tömb számára kevés helyet foglaltunk, de az is, hogy feleslegesen sokat.
- A megoldás: tömb helyett pointert deklarálunk, és ha tudjuk a kívánt méretet, memóriát már a megfelelő számú elemnek foglalunk.

Dinamikus tömb típus

- Mivel a pointert tömbként kezelhetjük, a programban kódjában ez semmilyen más változást nem eredményez:

```
int tomb[MAX];
…

for(i=0; i<n; ++i) {
    tomb[i]=i;
}
…

…
```

```
int *tomb;
…
tomb=malloc(n*sizeof(int));
for(i=0; i<n; ++i) {
    tomb[i]=i;
}
…
free(tomb);
…
```

Flexibilis tömb típus

- A C nyelvben lehetőség van arra, hogy egy pointer számára már lefoglalt memóriaterület méretét megváltoztassuk. A

```
void *realloc(void *ptr, size_t size)
```

  függvény a **`ptr`** által mutatott területet méretezi (és ha kell mozgatja) át.
- Ez sem használható viszont, ha a memória

Flexibilis tömb típus

- Egy megoldás a problémára a flexibilis tömb adattípus, ami a dinamikus tömb általánosítása.
- Ennek van olyan művelete amivel az indextípus felső határát módosíthatjuk, ezáltal változó elemszámú sorozatokat kezelhetünk, továbbá a megvalósítása kis méretű tömbökkel dolgozik.
- Adott **E** elemtípus esetén a flexibilis tömb (FTömb) adattípus értékhalmaza az összes

  A : 0..N ---> E

Flexibilis tömb műveletei

- Kiolvas(-> A:FTömb; -> i:int; <- X:E)
  - Az A függvény értékének kiolvasása
- Módosít(<-> A:FTömb; -> i:int; -> X:E)
  - Az A függvény értékének módosítása
- X=Y értékadás, ha X és Y FTömb típusú változók.
- Létesít(<-> A:FTömb; -> N:int)
  - N elemű flexibilis tömböt létesít.
- Megszüntet(<-> A:FTömb)
  - Törli az A flexibilis tömbhöz foglalt memóriát.

Flexibilis tömb műveletei

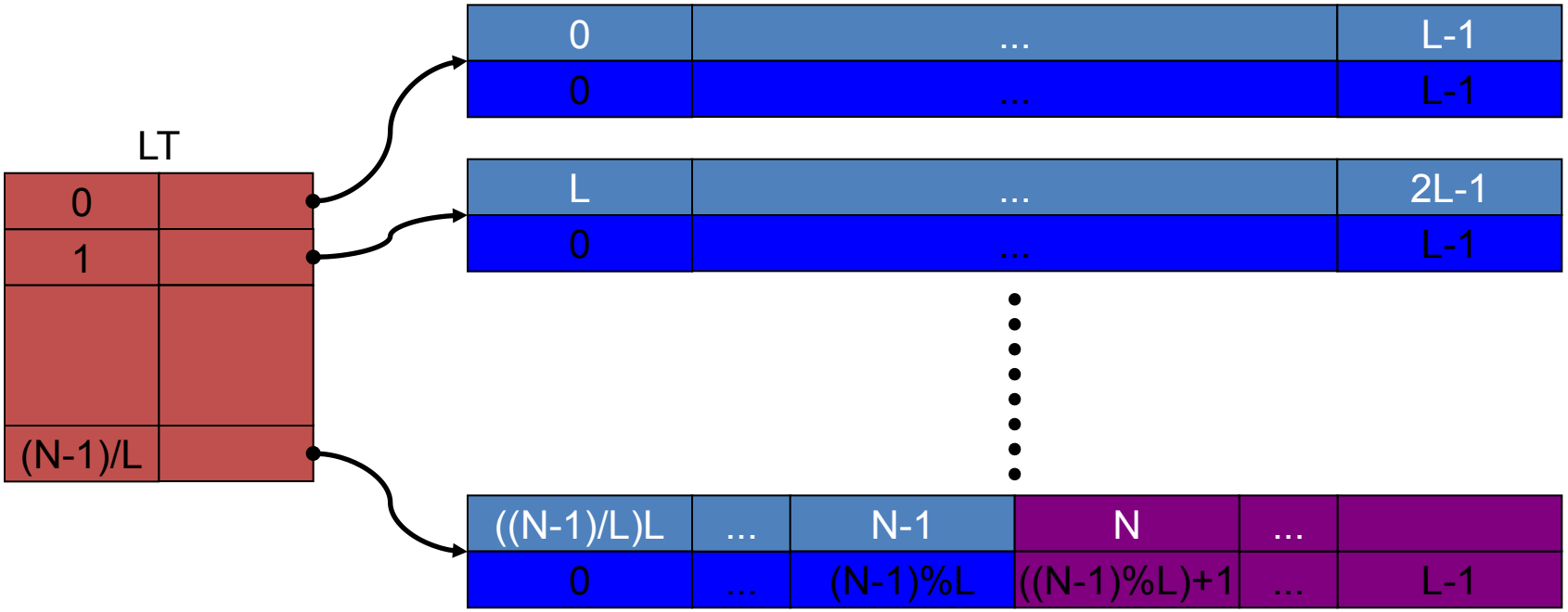
- Felső(-> A:FTömb): int
  - A felső határ lekérdezése
- Növel(-> A : FTömb; d:int)
  - Az aktuális indextípus felső határát a d értékkel növeli.
- Csökkent(-> A : FTömb; d:int)
  - Az aktuális indextípus felső határát a d értékkel csökkenti.

Flexibilis tömb megvalósítása

- A megvalósításhoz válasszunk egy **`L`** konstanst.
- **`L`** elemszámú dinamikussá tett tömbökből, amiket lapoknak nevezünk, állítsuk össze a nagy tömböt úgy, hogy felveszünk egy **`LT`** laptérkép tömböt, amelynek elemei lapokra mutató pointerek.

Flexibilis tömb megvalósítása

- Ezt szemlélteti az ábra.



Flexibilis tömb megvalósítása

```
/* Globális elemek a flexibilis tömb megvalósításához */

#define L ???                                        /* lapméret */

typedef enum {false, true} bool;              /* logikai típus */

typedef ???      elemtip;                  /* a tömb elemtípusa */
typedef elemtip *laptip;
typedef laptip  *lapterkeptip;

typedef struct ftomb {
        lapterkeptip lt;                          /* laptérkép */
        unsigned int hatar;             /* aktuális indexhatár */
} ftomb;
```

>>>

Flexibilis tömb megvalósítása

```
/* A műveletek megvalósítása:                                    */

void kiolvas(ftomb a, unsigned int i, elemtip *x)
{
    if(i < a.hatar) {
        *x = a.lt[i / L][i % L];
    }
}

void modosit(ftomb a, unsigned int i, elemtip x)
{
   if(i < a.hatar) {
      a.lt[i / L][i % L] = x;
   }
}
```

>>>

Flexibilis tömb megvalósítása

```
void letesit(ftomb *a, unsigned int n)
{
   int j;
   if(n) {
      a->hatar = n;
      a->lt=(elemtip**)malloc(
                  (1+((n-1)/L))*sizeof(lapterkeptip));
      for(j=0; j<=((n-1) / L); ++j) {
         a->lt[j]=(elemtip*)malloc(L*sizeof(elemtip));
                                        /* lapok létesítése */
      }
   } else {
      a->hatar = 0;
      a->lt = NULL;
   }
}
```

>>>

Flexibilis tömb megvalósítása

```
void megszuntet(ftomb *a)
{
   int j;
   if(a->hatar) {
      for(j=0; j<=((a->hatar-1) / L); ++j) {
         free(a->lt[j]);                    /* lapok törlése */
      }
      free(a->lt);
      a->hatar=0;
   }
}

unsigned int felso(ftomb a)
{
   return a.hatar;
}
```

Flexibilis tömb megvalósítása

```
void novel(ftomb *a, unsigned int d)
{
   int j;
   a->lt = (elemtip**)realloc(a->lt,
               (1+((a->hatar+d-1)/L))*sizeof(lapterkeptip));
   for(j=(a->hatar ? ((a->hatar-1)/L)+1 : 0) ;
                                 j<=(a->hatar+d-1)/L; ++j) {
      a->lt[j]=(elemtip*)malloc(L*sizeof(elemtip));
   }
   a->hatar += d;
}
```

>>>

Flexibilis tömb megvalósítása

```
void csokkent(ftomb *a, unsigned int d)
{
   int j;
   if(d <= a->hatar) {
      for(j=(a->hatar-d-1)/L +1; j<=(a->hatar-1)/L; ++j) {
         free(a->lt[j]);                  /* lapok törlése */
      }
      a->hatar -= d;
      a->lt = (elemtip**)realloc(a->lt,
                 (1+((a->hatar-1)/L))*sizeof(lapterkeptip));
    }
}
```

Rendezés több szempont szerint

- Problémafelvetés
  - A beolvasott adatokat rendezzük több szempont szerint is egy egyszerű rendezési algoritmussal és minden rendezés után legyen kiíratás is.
- Specifikáció
  - Flexibilis tömbbel dolgozzunk
  - Input
    - A tömb elemei.
  - Output
    - A különböző szempontok szerint rendezett tömb.

Rendezés több szempont szerint

- Algoritmustervezés:
  - A fő algoritmusban csak az elemeket kell beolvasni egy végjelig, majd rendre aktivizálni kell a különböző szempontok szerinti rendezést, végül az eredményt kiíratni.
  - A rendezés a beszúrórendezés lesz.

Beszúró rendezés

- Problémafelvetés
  - Rendezzük egy tömb elemeit
- Specifikáció
  - Input
    - Egy tömb melynek elemtípusán értelmezett egy rendezési reláció.
  - Output
    - A reláció alapján rendezett tömb.

Beszúró rendezés

- Algoritmustervezés:
  - A tömböt logikailag egy már rendezett és egy még rendezetlen részre osztjuk, és a rendezetlen rész első elemét beszúrjuk a rendezett elemek közé úgy, hogy azok rendezettek maradjanak.

Rendezés több szempont szerint

```
/* Rendezzük névsorba illetve átlag szerint a hallgatókat!
 * Flexibilis tömbbel történik a megvalósítás, tehát a
 * névsor hosszát nem kell előre megmondani.
 * Készítette: Dévényi Károly, devenyi@inf.u-szeged.hu
 *                1998. Február 16.
 * Módosította: Gergely Tamás, gertom@inf.u-szeged.hu
 *                2006. Augusztus 15.
 */

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

#define L 10                                    /* lapméret */
```

>>>

Rendezés több szempont szerint

```
typedef enum {false, true} bool;          /* logikai típus */

typedef struct elemtip {              /* a tömb elemtípusa */
                char nev[21];
                float adat;
} elemtip;

typedef elemtip *laptip;

typedef laptip  *lapterkeptip;

typedef struct ftomb {
        lapterkeptip lt;                        /* laptérkép */
        unsigned int hatar;          /* aktuális indexhatár */
} ftomb;

typedef bool (*RendRelTip)(elemtip, elemtip);
```

>>>

Rendezés több szempont szerint

```
/* A műveletek megvalósítása:                                    */

void kiolvas(ftomb a, unsigned int i, elemtip *x)
{
    if(i < a.hatar) {
        *x = a.lt[i / L][i % L];
    }
}

void modosit(ftomb a, unsigned int i, elemtip x)
{
   if(i < a.hatar) {
      a.lt[i / L][i % L] = x;
   }
}
```

>>>

Rendezés több szempont szerint

```
void letesit(ftomb *a, unsigned int n)
{
   int j;
   if(n) {
      a->hatar = n;
      a->lt=(elemtip**)malloc(
                  (1+((n-1)/L))*sizeof(lapterkeptip));
      for(j=0; j<=((n-1) / L); ++j) {
         a->lt[j]=(elemtip*)malloc(L*sizeof(elemtip));
                                        /* lapok létesítése */
      }
   } else {
      a->hatar = 0;
      a->lt = NULL;
   }
}
```

>>>

Rendezés több szempont szerint

```
void megszuntet(ftomb *a)
{
   int j;
   if(a->hatar) {
      for(j=0; j<=((a->hatar-1) / L); ++j) {
         free(a->lt[j]);                    /* lapok törlése */
      }
      free(a->lt);
      a->hatar=0;
   }
}

unsigned int felso(ftomb a)
{
   return a.hatar;
}
```

Rendezés több szempont szerint

```
void novel(ftomb *a, unsigned int d)
{
   int j;
   a->lt = (elemtip**)realloc(a->lt,
               (1+((a->hatar+d-1)/L))*sizeof(lapterkeptip));
   for(j=(a->hatar ? ((a->hatar-1)/L)+1 : 0) ;
                                 j<=(a->hatar+d-1)/L; ++j) {
      a->lt[j]=(elemtip*)malloc(L*sizeof(elemtip));
   }
   a->hatar += d;
}
```

>>>

Rendezés több szempont szerint

```
void csokkent(ftomb *a, unsigned int d)
{
   int j;
   if(d <= a->hatar) {
      for(j=(a->hatar-d-1)/L +1; j<=(a->hatar-1)/L; ++j) {
         free(a->lt[j]);                  /* lapok törlése */
      }
      a->hatar -= d;
      a->lt = (elemtip**)realloc(a->lt,
                 (1+((a->hatar-1)/L))*sizeof(lapterkeptip));
    }
}
```

>>>

Rendezés több szempont szerint

```
void beszuroRend(ftomb t, RendRelTip kisebb)
{  /* A Kisebb rendezési reláció szerinti helyben rendezés */
    int i,j;
    elemtip  e,f;
    for(i = 1; i < felso(t); ++i) {
        kiolvas(t, i, &e);
        j = i-1;
        while(true) {
            if(j<0)
                break;
            kiolvas(t, j, &f);
            if(kisebb(f, e))
                break;
            modosit(t, ((j--)+1), f);
        }
        modosit(t, j+1, e);
    }
} /* BeszuroRend */
```

>>>

Rendezés több szempont szerint

```
bool NevSzerint(elemtip X, elemtip Y)
{                  /* a névsor szerinti rendezési reláció  */
    return strcmp(X.nev, Y.nev) <= 0;
}

bool AdatSzerint(elemtip X, elemtip Y)
{                   /* az adat szerinti rendezési reláció  */
    return X.adat <= Y.adat;
}

bool CsokAdatSzerint(elemtip X, elemtip Y)
{          /* az adat szerint csökkenő rendezési reláció  */
    return X.adat >= Y.adat;
}
```

>>>

Rendezés több szempont szerint

```
int main()
{
    ftomb sor;
    elemtip hallg;                          /* beolvasáshoz */
    int i;
    letesit(&sor, 0);     /* a flexibilis tömb létesítése */
                                            /* beolvasás */
    printf("Kérem az adatsort, külön sorban név és adat!\n");
    printf("A végét a * jelzi.\n");
    scanf("%20[^\n]%*[^\n]", hallg.nev); getchar();
    i = 0;                  /* az i. helyre fogunk beírni */
    while(strcmp(hallg.nev, "*")) {
        novel(&sor, 1);     /* a flexibilis tömb bővítése */
        scanf("%f%*[^\n]", &hallg.adat); getchar();
        modosit(sor, i++, hallg);
        scanf("%20[^\n]%*[^\n]", hallg.nev); getchar();
    }
```

Rendezés több szempont szerint

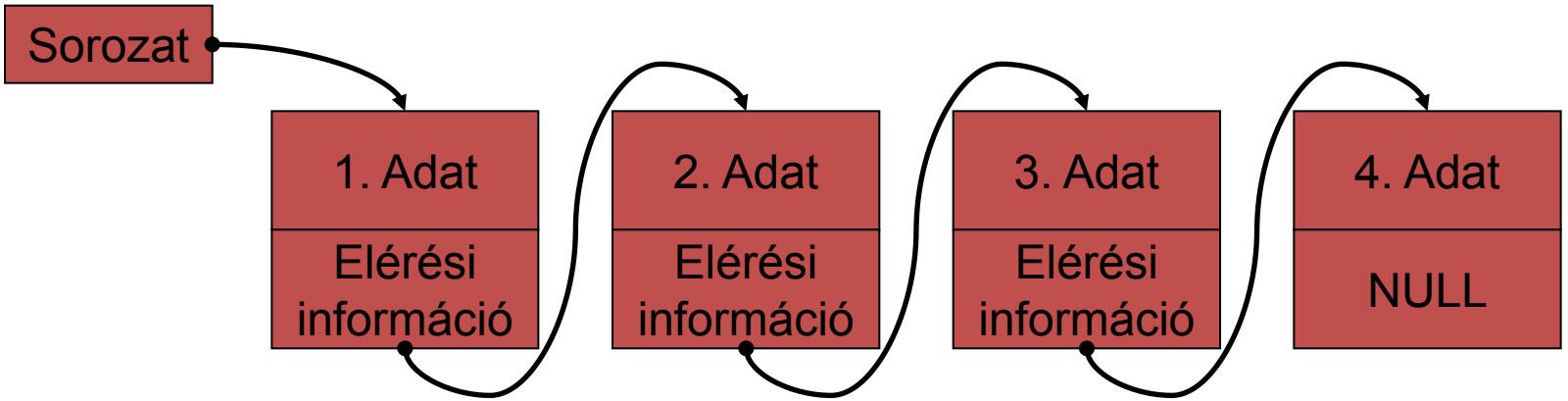
```
    beszuroRend(sor, NevSzerint); /* Rend. névsor szerint */
    printf("Névsor szerint rendezve:\n");
    for(i = 0; i < felso(sor); ++i) {          /* Kiíratás */
        kiolvas(sor, i, &hallg);
        printf("%6.2f %s\n", hallg.adat, hallg.nev);
    }
    beszuroRend(sor, AdatSzerint);  /* Rend. adat szerint */
    printf("Adat szerint rendezve:\n");
    for(i = 0; i < felso(sor); ++i) {          /* Kiíratás */
        kiolvas(sor, i, &hallg);
        printf("%6.2f %s\n", hallg.adat, hallg.nev);
    }
    beszuroRend(sor, CsokAdatSzerint);   /* Rendezés újra */
    printf("Adat szerint csökkenő sorba rendezve:\n");
    for(i = 0; i < felso(sor); ++i) {          /* Kiíratás */
        kiolvas(sor, i, &hallg);
        printf("%6.2f %s\n", hallg.adat, hallg.nev);
    }
    megszuntet(&sor);         /* a flexibilis tömb törlése */
}
```

Láncok

- Tekintsük azonos típusú adatelemek sorozatát.
- Ha a sorozat bármely pozíciójára vonatkozó bővítés és törlés műveletet is akarunk végezni, akkor a tömbös reprezentálás nem lehet hatékony, ehelyett a sorozatnak láncolással történő reprezentálása ajánlott.
- Láncoláson olyan adatszerkezetet értünk, amely a tárolandó sorozat minden elemét egy olyan rekordban (cellában) tárolja, amely a sorozat következő elemének elérését biztosító információt is tartalmazza.

Láncok

- Az elérési információ lehet egy pointer érték, amely a sorozat következő elemét tartalmazó cellára (dinamikus változóra) mutató pointer.
- A sorozatot az első elemére mutató pointerrel adhatjuk meg.



Láncok

- C-ben így hozhatunk létre láncolt lista típust:

```
typedef ??? elemtip;    /* a sorozat elemtípusa */

typedef struct cellatip {
    elemtip adat;                    /* adatelem */
    struct cellatip *csat;
                    /* a következő elem cellájára */
} cellatip;

typedef struct cellatip *pozicio;    /* pointer */
```

Láncok

- Deklaráljuk a **`p`** pointert! Lehet a következő módokon:

```
cellatip *p;
struct cellatip *p;
pozicio p;
```

- A **`p`** pointer által megmutatott cella egyes mezőire így hivatkozhatunk:
  - **`p->adat`**
  - **`p->csat`**

Láncok

- A **`->`** és **`.`** struktúraoperátorok a precedencia-hierarchia csúcsán állnak, és ezért nagyon szorosan kötnek.
- A

  ```
  ++p->adat
  ```

  nem **`p`**-t, hanem az **`adat`** mezőt inkrementálja, mivel az alapértelmezés szerinti zárójelezés:

  ```
  ++(p->adat)
  ```

Láncok

- Zárójelek használatával a kötés megváltoztatható:

  `(++p)->adat`

  az **`adat`**-hoz való hozzáférés előtt növeli **`p`**-t, míg

  `(p++)->adat`

  azt követően inkrementál.
- Láncok esetén vigyázzunk ezekkel a műveletekkel, mert egyáltalán nem biztos, hogy két **`cellatip`** típusú láncelem közvetlenül egymás után helyezkedik el a memóriában!

Láncok

- A lánc soron következő elemét a

  `p->csat`

  pointeren keresztül a

  `*p->csat`

  hozza be.

Láncok

```
*p->csat++
```

azután inkrementálja **`csat`**-t, miután hozzáfért ahhoz, amire mutat (ekkor a lánc megszakadhat!)

```
(*p->csat)++
```

azt növelné, amire csat mutat, (de ezt most nem lehet, mert ez egy **`struct`**)

```
*p++->csat
```

azután inkrementálja **`p`**-t, hogy hozzáfért ahhoz, amire csat mutat (de ekkor nem biztos, hogy **`p`** továbbra is a lánc valamelyik elemére mutat).

Láncok

- Láncok bejárására írhatunk olyan függvényt amelynek paramétere az elvégzendő művelet:

```
typedef void (*muvelettip)(elemtip*);

void bejar(pozicio lanc, muvelettip muv)
{
   pozicio p;
   for(p = lanc; p != NULL; p = p->csat) {
      /* művelet a sorozat elemén */
      muv(&(p->adat));
   }
}
```

Kérdések, válaszok