

Programozás Alapjai

Dr. Gergely Tamás
Dr. Jász Judit

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2020

(v0908)



- 1 **Bemutakozás**
- Kurzus információk
 - A SZTE és az informatikai képzés

- 2 **Linux**
- Alapfogalmak
 - Linux parancsok
 - Linux shell
 - Felhasználók
 - Hálózat

- 3 **Gyors C áttekintés**
- Bevezető
 - Pénzváltás (1. verzió)
 - Pénzváltás (2. verzió)
 - Röppálya számítás
 - Röppálya szimuláció
 - Az év napja
 - Csúszóátlag adott elemszámmra
 - Csúszóátlag parancssorból
 - Basename standard inputról
 - Basename parancssorból
 - Tér legtávolabbi pontjai
 - A nappalis gyakorlat értékelése

- 4 **Alapok**
- Alapfogalmak
 - A programozás fázisai
 - Algoritmus vezérlése
 - A C nyelvű program
 - Szintaxis
 - A C nyelv elemi adattípusai
 - A C nyelv utasításai

- 5 **Vezérlési szerkezetek**
- Bevezetés
 - Szekvenciális vezérlés
 - Függvények
 - Szelekciós vezérlések
 - Ismétléses vezérlések 1.
 - Eljárásvezérlés
 - Ismétléses vezérlések 2.

- 6 **Folyamatábra és struktúradiagram**

- 7 **Adatszerkezetek**
- Az adatkezelés szintjei
 - Elemi adattípusok
 - Pointer adattípus
 - Tömb adattípus

- Sztringek
- Pointerek és tömbök C-ben
- Rekord adattípus
- Függvény pointer
- Halmaz adattípus
- Flexibilis tömbök
- Láncolt listák
- Típusokról C-ben

- 8 **IO**
- Alapok
 - Adatállományok

- 9 **C fordítás**
- A fordítás folyamata
 - A preprocesszor
 - A C fordító
 - Assembler
 - Linker és modulok

- 10 **Gyakorlati kérdések**
- Memóriahasználat
 - Gyakori C hibák
 - where.c felboncolva

- 1 **Bemutakozás**
 - Kurzus információk
 - A SZTE és az informatikai képzés
- 2 **Linux**
 - Alapfogalmak
 - Linux parancsok
 - Linux shell
 - Felhasználók
 - Hálózat
- 3 **Gyors C áttekintés**
 - Bevezető
 - Pénzváltás (1. verzió)
 - Pénzváltás (2. verzió)
 - Röppálya számítás
 - Röppálya szimuláció
 - Az év napja
 - Csúszóátlag adott elemszámmra
 - Csúszóátlag parancssorból
 - Basename standard inputról
 - Basename parancssorból
 - Tér legtávolabbi pontjai
 - A nappalis gyakorlat értékelése

- 4 **Alapok**
 - Alapfogalmak
 - A programozás fázisai
 - Algoritmus vezérlése
 - A C nyelvű program
 - Szintaxis
 - A C nyelv elemi adattípusai
 - A C nyelv utasításai
- 5 **Vezérlési szerkezetek**
 - Bevezetés
 - Szekvenciális vezérlés
 - Függvények
 - Szelekciós vezérlések
 - Ismétléses vezérlések 1.
 - Eljárásvezérlés
 - Ismétléses vezérlések 2.
- 6 **Folyamatábra és struktúradiagram**
- 7 **Adatszerkezetek**
 - Az adatkezelés szintjei
 - Elemi adattípusok
 - Pointer adattípus
 - Tömb adattípus

- 8 **IO**
 - Alapok
 - Adatállományok
- 9 **C fordítás**
 - A fordítás folyamata
 - A preprocesszor
 - A C fordító
 - Assembler
 - Linker és modulok
- 10 **Gyakorlati kérdések**
 - Memóriaahasználat
 - Gyakori C hibák
 - where.c felboncolva

Adattípus

Az adattípus a programnak egy olyan komponense, amely két összetevője, az értékhalmaz és az értékhal-maz elemein végezhető műveletek által meghatáro-zott.

- Minden adattípus vagy elemi, vagy más adattípusokból képzett összetett adattípus.



Az adatkezelés szintjei

- Számítógépes problémamegoldás során az adatkezelésnek különböző szintjeit lehet elkülöníteni.
 - Ezek közötti világos különbségtételnek fontos szerepe van az algoritmustervezésben.
- Az adatkezelés szintjei egyúttal különböző absztrakciós szinteknek felelnek meg.
 - Probléma szintje
 - Szaktudományos és matematikai szint
 - Absztrakt adattípus szint
 - Virtuális adattípus szint
 - Fizikai adattípus szint



Az adatkezelés szintjei

Probléma szintje.

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



Az adatkezelés szintjei

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.



Az adatkezelés szintjei

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.

Az adatkezelés szintjei

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.



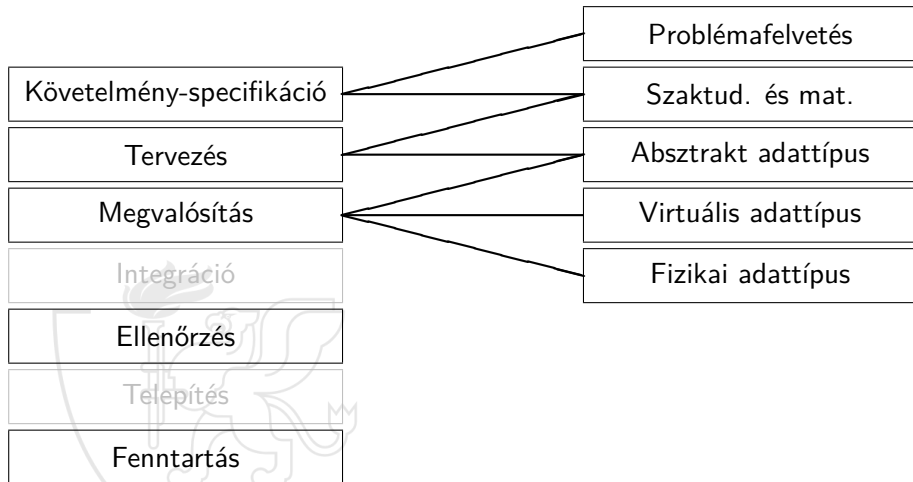
Az adatkezelés szintjei

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 bájtok (bitek) véges sorozata, tehát végső soron minden adattípust e lineáris szerkezetű tárolóval kell reprezentálni.



A vízésés modell és az adatkezelési szintek kapcsolata



- 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.
- 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.

- 1 **Bemutakozás**
- Kurzus információk
 - A SZTE és az informatikai képzés

- 2 **Linux**
- Alapfogalmak
 - Linux parancsok
 - Linux shell
 - Felhasználók
 - Hálózat

- 3 **Gyors C áttekintés**
- Bevezető
 - Pénzváltás (1. verzió)
 - Pénzváltás (2. verzió)
 - Röppálya számítás
 - Röppálya szimuláció
 - Az év napja
 - Csúszóátlag adott elemszámmra
 - Csúszóátlag parancssorból
 - Basename standard inputról
 - Basename parancssorból
 - Tér legtávolabbi pontjai
 - A nappalis gyakorlat értékelése

- 4 **Alapok**
- Alapfogalmak
 - A programozás fázisai
 - Algoritmus vezérlése
 - A C nyelvű program
 - Szintaxis
 - A C nyelv elemi adattípusai
 - A C nyelv utasításai

- 5 **Vezérlési szerkezetek**
- Bevezetés
 - Szekvenciális vezérlés
 - Függvények
 - Szelekciós vezérlések
 - Ismétléses vezérlések 1.
 - Eljárásvezérlés
 - Ismétléses vezérlések 2.

- 6 **Folyamatábra és struktúradiagram**

- 7 **Adatszerkezetek**
- Az adatkezelés szintjei
 - **Elemi adattípusok**
 - Pointer adattípus
 - Tömb adattípus

- Sztringek
- Pointerek és tömbök C-ben
- Rekord adattípus
- Függvény pointer
- Halmaz adattípus
- Flexibilis tömbök
- Láncolt listák
- Típusokról C-ben

- 8 **IO**
- Alapok
 - Adatállományok

- 9 **C fordítás**
- A fordítás folyamata
 - A preprocesszor
 - A C fordító
 - Assembler
 - Linker és modulok

- 10 **Gyakorlati kérdések**
- Memóriahasználat
 - Gyakori C hibák
 - where.c felboncolva

- Az adattípusokat osztályozhatjuk aszerint is, hogy az értékalmazuk elemei összetettek-e vagy elemiek.
- Az elemi adattípusok értékeit nem lehet önmagukban értelmes részekre bontani.
- A C nyelv elemi adattípusai.
 - Egész típusok (`int`, `signed`, `unsigned`, `short`, `long`).
 - Karakter típus (`char`, `signed`, `unsigned`).
 - Felsorolás (`enum` típusképzéssel).
 - Logikai típus (C-ben nincs közvetlen megvalósítása).
 - Valós típusok (`float`, `double`, `long double`).
- Ha a nyelv szintaktikája szerint a program egy adott pontján típusnak kellene következnie de az hiányzik, a fordító a típus helyére automatikusan `int`-et helyettesít.

A C nyelv elemi adattípusai

32 bites architektúrán

C típus	méret (bájt/bit)	alsó határ	felső határ
char	1 / 8	?	?
signed char	1 / 8	-128	127
unsigned char	1 / 8	0	255
short int	2 / 16	-32 768	32 767
signed short int	2 / 16	-32 768	32 767
unsigned short int	2 / 16	0	65 535
int	4 / 32	-2 147 483 648	2 147 483 647
signed int	4 / 32	-2 147 483 648	2 147 483 647
unsigned int	4 / 32	0	4 294 967 295
long int	4 / 32	-2 147 483 648	2 147 483 647
signed long int	4 / 32	-2 147 483 648	2 147 483 647
unsigned long int	4 / 32	0	4 294 967 295
long long int	8 / 64	-2 ⁶³	2 ⁶³ -1
signed long long int	8 / 64	-2 ⁶³	2 ⁶³ -1
unsigned long long int	8 / 64	0	2 ⁶⁴ -1
float	4 / 32	-3.4028234663852886E+38	3.4028234663852886E+38
double	8 / 64	-1.7976931348623157E+308	1.7976931348623157E+308
long double	8 / 64	-1.7976931348623157E+308	1.7976931348623157E+308

Egész típusok a C nyelvben

Típus és módosítók

- A C nyelvben az egész típus az `int`.
- Az `int` típus értékkészlete az alábbi kulcsszavakkal módosítható:
 - `signed` A típus előjeles értékeket fog tartalmazni (`int`, `char`).
 - `unsigned` A típus csak előjeltelen, nemnegatív értékeket fog tartalmazni (`int`, `char`).
 - `short` Rövidebb helyen tárolódik, így kisebb lesz az értékkészlet (`int`).
 - `long` Hosszabb helyen tárolódik, így bővebb lesz az értékkészlet (`int`). Duplán is alkalmazható (`long long`).
- Az egész típusok az értékkészlet határain belüli minden egész értéket pontosan ábrázolnak.
- Az egyes gépeken az egyes típusok mérete más-más lehet, de minden C megvalósításban teljesülnie kell a `sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)` relációnak.

Egész típusok a C nyelvben

Műveletek

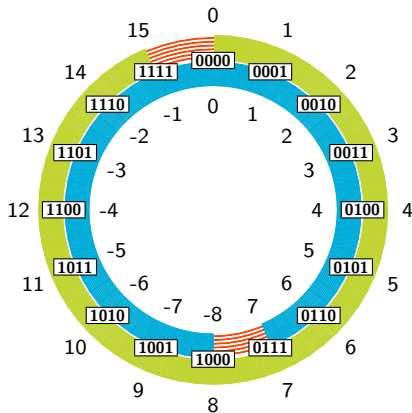
- A C nyelv különféle egész adattípusai az értékhalmazukban különböznek egymástól, az értelmezett műveletükben megegyeznek.
 - Az egész adattípusokon általában az 5 matematikai alpműveletet és az értékadás műveletét értelmezzük, de C nyelven ennél jóval többet.
- Egész kifejezésben bármely egész típusú tényező (akár vegyesen többféle is) szerepelhet.
- Egész literál típusa az a legalább `int` méretű egész típus, amely a legszűkebb olyan értékhalmazú, amelynek eleme a kifejezés értéke, vagy a számleírásban megadott típus (`l`, `u` suffix).
- Értékadó művelet jobb oldalán álló kifejezés kiértékelése független attól, hogy a bal oldalon milyen típusú változó van.
- A `/` művelet két egész értékre alkalmazva maradékos osztást jelent!

- Egy n bites tárterületnek 2^n állapota van, vagyis egy n biten tárolt adattípusnak legfeljebb ennyi különböző értéke lehet.
- Egész típusoknál az értékhalmoz szokásos leképezése a következő:
 - Ha negatív számok nem szerepelnek az értékhalmozban, akkor az értékhalmoz a $[0 \dots 2^n - 1]$ zárt intervallum. Az n egymás utáni bitet bináris számként értelmezve kapjuk meg a reprezentált értéket, illetve az értéket bináris számként felírva és n számjegyig balról 0 számjegyekkel kiegészítve kapjuk a számot leíró bitsorozatot.
 - Ha az értékhalmozban negatív számok is szerepelnek, akkor az értékhalmoz a $[-2^{n-1} \dots 2^{n-1} - 1]$ zárt intervallum. A $[0 \dots 2^{n-1} - 1]$ értékek tárolása és a 0 értékű bittel kezdődő bitsorozat értelmezése megegyezik a fentiekkel. Egy $x \in [-2^{n-1} \dots - 1]$ érték tárolása úgy történik, mintha az $x + 2^n$ értéket próbálnánk nemnegatív számként n biten tárolni, illetve az 1-essel kezdődő bitsorozat pozitív számként értelmezett értékéből 2^n -t levonva kapjuk a reprezentált negatív értéket.

Egész típusok a C nyelvben

Tárolás [2/3]

- Ha például $n = 4$ bites egész típussal dolgoznánk, akkor összesen $2^4 = 16$ értéket tudnánk megkülönböztetni:
 - Előjeltelen esetben az értékhalmoz a $[0 \dots 15]$ zárt intervallum lenne.
 - Előjeles esetben a $[-8 \dots 7]$ értékeket tudnánk ábrázolni.
 - Az előjeltelen $[8 \dots 15]$ értékek fizikailag pontosan ugyanúgy tárolódnak, mint az előjeles $[-8 \dots -1]$ értékek.



- A negatív számoknak ezt a tárolási módját kettes komplementnek hívjuk.
 - Egy x érték kettes komplemente a $\sim x + 1$ érték, vagyis x minden bitjét negáljuk, majd hozzáadunk egyet.
- A kettes komplement ábrázolás előnyei:
 - Gépi szinten az előjeles és előjeltelen típusokkal való elemi műveleteket nem kell megkülönböztetni, ugyanaz a logika működik mindkettőre.
 - Nem kell műveletet cserélni negatív érték hozzáadásakor vagy kivonásakor, vagyis $x + -c$ -ből nem kell $x - c$ -t csinálni.



Típusok határértékei

limits.h

- A limits.h-ban sok konstans van deklarálva, ha ezeket használni szeretnénk, akkor a #include <limits.h> sort kell beszúrni valahová a program elejére.
- Néhány definíció (32 bites rendszeren):

```
#define SCHAR_MIN    (-128)           /* signed char */
#define SCHAR_MAX    127             /* signed char */
#define UCHAR_MAX    255             /* unsigned char */
#define SHRT_MIN     (-32768)        /* short int */
#define SHRT_MAX     32767           /* short int */
#define USHRT_MAX    65535          /* unsigned short int */
#define INT_MIN      (-INT_MAX - 1)  /* int */
#define INT_MAX      2147483647      /* int */
#define UINT_MAX     4294967295U     /* unsigned int */
#define LONG_MAX     2147483647L     /* long int */
#define LONG_MIN     (-LONG_MAX - 1L) /* long int */
#define ULONG_MAX    4294967295UL    /* unsigned long int */
#define LLONG_MAX    9223372036854775807LL /* long long int */
#define LLONG_MIN    (-LLONG_MAX - 1LL) /* long long int */
#define ULLONG_MAX   18446744073709551615ULL /* unsigned long 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 alapvetően karakterek (betűk, számjegyek, írásjelek) tárolására való.
 - 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

Literálok

- Egy C programban karakter értékeket megadhatunk
 - karakterkóddal számértékként, vagy
 - aposztrófok (') közé írt karakterrel.
- A speciális karaktereket, illetve magát az aposztrófit (és végső soron tetszőleges karaktert is) escape-szekvenciákkal lehet megadni.
 - Az escape-szekvenciákat a \ (backslash) karakterrel kell kezdeni.
 - Néhány példa

újsor	lf	<code>\n</code>
vízszintes tab	ht	<code>\t</code>
backslash	<code>\</code>	<code>\\</code>
aposztróf	<code>'</code>	<code>\'</code>
Oddd kódú karakter		<code>\ddd</code>

kocsi-vissza	cr	<code>\r</code>
backspace	bs	<code>\b</code>
lapdobás	ff	<code>\f</code>
csengő	bell	<code>\a</code>
null karakter	null	<code>\0</code>

Karakter adattípus a C nyelven

Kódtáblázat

- Az, hogy egy adott karakterkódhoz milyen karakter tartozik, a használt kódtáblázat határozza meg.
- A kódtábláról csak annyit tételvezhetünk fel, hogy
 - 'a' < 'b' < ... < 'z'
 - 'A' < 'B' < ... < 'Z'
 - '0' + 1 == '1', ..., '8' + 1 == '9'
- A C nyelv alapvetően az ASCII kódtáblát használja (ez a 0 és 127 közötti kódú karaktereket rögzíti), de a megjelenített karakterkép nagyon sokmindentől függhet.
- A nem ASCII karakterekkel vigyázzunk, mert pl. UTF8 kódolás esetén egyes karakterek (pl. 'á' vagy 'ő') több bájtban tárolódnak, és ezt a C nyelv char típusa nem tudja lekezelni!
 - Bár a sztringeknél látszólag rendben lehet az extra karakterek kezelése, egy UTF8-as kódolású "árvíztűrő" sztring C-ben valójában nem 9+1, hanem 13+1 bájtot foglal.

Karakter adattípus a C nyelven

Karakter és karakterkód

- Mint említettük a `char` adattípus egészként is használható.
 - A konverzió a kétfajta megadott érték között automatikus, így például `'\ddd' == 0ddd`, vagyis ASCII kódtáblázat esetén például `'\060' == '0' == 48 == 060`, `'\101' == 'A' == 65 == 0101`, `'\172' == 'z' == 122 == 0172`.
- Konvertáljunk egy tetszőleges számjegy karaktert (`ch`) a neki megfelelő egész számmá és egy egyjegyű egészet (`i`) karakterré.

```
i = ch - '0';  
ch = i + '0';
```

- Konvertáljunk kisbetűt (`ch`) nagybetűvé (`CH`) és nagybetűt kisbetűvé.

```
CH = ch - 'a' + 'A';  
ch = CH - 'A' + 'a';
```

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



Az int (egész) adattípus műveletei

Bitenkénti logikai műveletek

- $\text{int} \rightarrow \text{int}$

(*egész* \rightarrow *egész*)

egy operandusú műveletek

~ bitenkénti
negáció

- $\text{int} \times \text{int} \rightarrow \text{int}$

(*egész* \times *egész* \rightarrow *egész*)

két operandusú műveletek

& bitenkénti és
| bitenkénti vagy
^ bitenkénti kizáró
vagy
<< balra léptetés
>> jobbra léptetés

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

Bitenkénti logikai műveletek

\sim – negáció – egyes komplementens

- A bitenkénti negáció (\sim) az érték minden bitjét az ellenkezőjére változtatja.

```
 $\sim 0x16 = 0xe9$ 
```

```
 $\sim 00010110 == 11101001$ 
```

- Általában maszkok kialakítására szokás használni:

```
MASK =  $\sim 3$ 
```

A MASK utolsó két bitje 0 lesz, a konkrét értéke pedig a 3 aktuális típusától függ.

Bitenkénti logikai műveletek

& – és – AND

- A bitenkénti és művelet (&) az eredmény i . bitjét a két operandusának i . bitjei alapján a logikai és művelet szerint állítja be. A 0 bit hamis, az 1 bit igaz.

```
0x16 & 0x3a = 0x12
```

```
00010110  &
```

```
00111010 ==
```

```
00010010
```

- Általában maszkolásra, egyes bitek 0-ra állítására szokás használni:

```
(v & 1) == 1
```

Az eredménynek legfeljebb az utolsó bitje lehet 1, ha a v utolsó bitje 1 volt, vagyis v páratlan volt.

Bitenkénti logikai műveletek

| – *megengedő vagy* – *inclusive OR*

- A bitenkénti vagy művelet (|) az eredmény i . bitjét a két operandusának i . bitjei alapján a logikai vagy művelet szerint állítja be. A 0 bit hamis, az 1 bit igaz.

```
0x16 | 0x3a = 0x3e
```

```
00010110 |  
00111010 ==  
00111110
```

- Általában egyes bitek 1-re állítására szokás használni:

```
v |= MASK
```

A v változó azon bitjeit, melyek a MASK-ban 1-esek voltak szintén 1-re állítja, a többi nem változtatja meg.

Bitenkénti logikai műveletek

\wedge – *kizáró vagy* – *exclusive OR*

- A bitenkénti *kizáró vagy* művelet (\wedge) az eredmény i . bitjét a két operandusának i . bitjei alapján a logikai *kizáró vagy* művelet szerint állítja be. A 0 bit hamis, az 1 bit igaz.

```
0x16 ^ 0x3a = 0x2c
```

```
00010110 ^
```

```
00111010 ==
```

```
00101100
```

- Általában bitek összehasonlítására, átbillentésére szokás használni:

```
v ^= MASK
```

A v változó azon bitjeit, melyek a MASK-ban 1-esek voltak megváltoztatja, a többi nem.

Bitenkénti logikai műveletek

\ll – *balra léptetés* – *left shift*

- A balra léptetés (\ll) a baloldali operandus bitjeit a jobboldali operandus értékével balra tolja. A jobboldalon 0 biteket szúr be.

```
0x36 << 4 == 0x60
```

```
0x96 << 4 == 0x60
```

```
00110110 << 4 == 01100000
```

```
10010110 << 4 == 01100000
```

- Gyakran használják kettő hatvánnyal való szorzásra vagy bit kiválasztására:

```
MASK = 1 << 8
```

A MASK-ban a 2^8 helyiértékű bit lesz 1-es, ami tulajdonképpen az 1×2^8 érték.

Bitenkénti logikai műveletek

>> – jobbra léptetés – right shift

- A jobbra léptetés (>>) a baloldali operandus bitjeit a jobboldali operandus értékével jobbra tolja. Előjeltelen értéknél a jobboldalon 0-kat, előjeles értéknél pedig a legfelső bit értékét szúrja be.

unsigned ...

```
0x36 >> 4 == 0x03  
0x96 >> 4 == 0x09
```

signed ...

```
0x36 >> 4 == 0x03  
0x96 >> 4 == 0xf9
```

```
00110110 >> 4 == 00000011
```

```
10010110 >> 4 == 00001001
```

```
00110110 >> 4 == 00000011
```

```
10010110 >> 4 == 11111001
```

- Gyakran használják kettő hatvánnyal való osztásra:

```
v >>= 8
```

A v értékét 2^8 -al osztjuk.

Bitenkénti logikai műveletek

Bitenkénti és logikai műveletek

- Gondosan meg kell különböztetnünk az `&` és `|` bitenkénti operátorokat az `&&` és `||` logikai műveletektől.
 - A bitműveletek az érték egészét nézve tulajdonképpen matematikai műveleteknek tekinthetők, amelyek mindkét operandust felhasználják.
 - Logikai műveletek esetén az operandusok csak szükség szerint balról jobbra értékelődnek ki.
 - Ráadásul az eredmény logikai értéke is különbözhet. Ha például `x` értéke 1 és `y` értéke 2, akkor

```
x & y == 0  
x && y != 0
```

Bitenkénti logikai műveletek

Előjeles vagy előjeltelen a char típus?

- Készítsünk egy programot, ami a jobbra léptetés művelet segítségével eldönti, hogy a char adattípust a fordító előjeles vagy előjeltelen típusként kezeli.
- char.c [1-14]:

```
1 /* A char pontos típusának ellenőrzése.
2  * 2006. Augusztus 8.  Gergely Tamás, gertom@inf.u-szeged.hu
3  */
4
5 #include <stdio.h>
6
7 int main() {
8     char a;
9     unsigned char b;
10    a = b = 128;
11    a >>= 1; b >>= 1;
12    printf("Ezen a gépen %ssigned char van\n", ((a == b) ? "un" : ""));
13    return 0;
14 }
```

Bitenkénti logikai műveletek

Példa [1/2]

- Tegyük fel, hogy több kis számot szeretnénk egyetlen `int` változóba beleírni. Például az év egy időpontjának hónapját, napját, óráját, percét és másodpercét rendre 4-5-5-6-6 biten.
- Az értékek tárolásához használhatunk két segédfüggvényt:

```
unsigned getBits(unsigned val, unsigned pos, unsigned len) {  
    return (val >> (pos + 1 - len)) & ~(~0 << len);  
}  
  
void setBits(unsigned val, unsigned pos, unsigned len, unsigned *ptr) {  
    unsigned MASK = ~(~(-0 << len) << (pos + 1 - len));  
    *ptr = (*ptr & MASK) | (val << (pos + 1 - len));  
}
```

- Ezekkel könnyen megvalósítható például az órák lekérdezése vagy a percek beállítása:

```
unsigned getHours(unsigned value) {  
    return getBits(value, 16, 5);  
}  
  
void setMinutes(unsigned value, unsigned *ptr) {  
    setBits(value, 11, 6, ptr);  
}
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return ( val >> (pos + 1 - len)) & ~(~ 0 << len);
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~(~(~ 0 << len) << (pos + 1 - len));
```

```
*ptr = ( *ptr & MASK ) | ( val << (pos + 1 - len));
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return ( val >> (pos + 1 - len)) & ~(~ 0 << len);  
return ( 27609578 >> ( 16 + 1 - 5 )) & ~(~ 0 << 5 );
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~(~(~ 0 << len) << (pos + 1 - len));
```

```
*ptr = ( *ptr & MASK ) | ( val << (pos + 1 - len));
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return ( val >> (pos + 1 - len)) & ~(~ 0 << len);
```

```
return ( 27609578 >> ( 16 + 1 - 5 )) & ~(~ 0 << 5 );
```

```
return (0x01a549ea >> ( 16 + 1 - 5 )) & ~(~0x00000000 << 5 );
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~(~(~ 0 << len) << (pos + 1 - len));
```

```
*ptr = ( *ptr & MASK ) | ( val << (pos + 1 - len));
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return ( 27609578 >> ( 16 + 1 - 5 )) & ~(~ 0 << 5 );
```

```
return (0x01a549ea >> ( 16 + 1 - 5 )) & ~(~0x00000000 << 5 );
```

```
return (0x01a549ea >> 12 ) & ~( 0xffffffff << 5 );
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~(~(~ 0 << len) << (pos + 1 - len));
```

```
*ptr = ( *ptr & MASK )|( val << (pos + 1 - len));
```


Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return (0x01a549ea >> ( 16 + 1 - 5 )) & ~(~0x00000000 << 5 );
```

```
return (0x01a549ea >> 12 ) & ~( 0xffffffff << 5 );
```

```
return 0x00001a54 & ~ 0xffffffe0 ;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~(~(0 << len) << (pos + 1 - len));
```

```
*ptr = ( *ptr & MASK )|( val << (pos + 1 - len));
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return (0x01a549ea >> 12) & ~(0xffffffff << 5);
```

```
return 0x00001a54 & ~0xfffffe0;
```

```
return 0x00001a54 & 0x0000001f;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~(~(0 << len) << (pos + 1 - len));
```

```
*ptr = (*ptr & MASK) | (val << (pos + 1 - len));
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 0x00001a54 & ~ 0xffffffff00000000 ;
```

```
return 0x00001a54 & 0x000000000000001f ;
```

```
return 0x000000014 ;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~(~(0 << len) << (pos + 1 - len));
```

```
*ptr = (*ptr & MASK) | (val << (pos + 1 - len));
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 0x00001a54 & 0x0000001f ;
```

```
return 0x00000014 ;
```

```
return 20 ;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~(~(0 << len) << (pos + 1 - len));
```

```
*ptr = (*ptr & MASK) | (val << (pos + 1 - len));
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 0x00000014;
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~(~(0 << len) << (pos + 1 - len));
```

```
*ptr = (*ptr & MASK) | (val << (pos + 1 - len));
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~(~(0 << len) << (pos + 1 - len));
```

```
*ptr = (*ptr & MASK) | (val << (pos + 1 - len));
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~(~(0 << len) << (pos + 1 - len));
```

```
unsigned MASK = ~(~(0 << 6) << (11 + 1 - 6));
```

```
*ptr = (*ptr & MASK) | (val << (pos + 1 - len));
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~(~(0 << len) << (pos + 1 - len));
```

```
unsigned MASK = ~(~(0 << 6) << (11 + 1 - 6));
```

```
unsigned MASK = ~(~(0x00000000 << 6) << (11 + 1 - 6));
```

```
*ptr = (*ptr & MASK) | (val << (pos + 1 - len));
```


Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~(~(0 << 6) << (11 + 1 - 6));
```

```
unsigned MASK = ~(~(0x00000000 << 6) << (11 + 1 - 6));
```

```
unsigned MASK = ~(~(0xffffffff << 6) << 6);
```

```
*ptr = (*ptr & MASK) | (val << (pos + 1 - len));
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~(~(0x00000000 << 6) << (11 + 1 - 6));
```

```
unsigned MASK = ~(~(0xffffffff << 6) << 6);
```

```
unsigned MASK = ~(~0xffffffffc0 << 6);
```

```
*ptr = (*ptr & MASK) | (val << (pos + 1 - len));
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~( ~( 0xffffffff << 6 ) << 6 );
```

```
unsigned MASK = ~( ~ 0xfffffc0 << 6 );
```

```
unsigned MASK = ~( 0x0000003f << 6 );
```

```
*ptr = ( *ptr & MASK ) | ( val << (pos + 1 - len));
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~(~ 0xffffffffc0 << 6);
```

```
unsigned MASK = ~( 0x0000003f << 6);
```

```
unsigned MASK = ~( 0x00000fc0);
```

```
*ptr = ( *ptr & MASK ) | ( val << (pos + 1 - len));
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~( 0x0000003f          <<          6          );
```

```
unsigned MASK = ~( 0x00000fc0          );
```

```
unsigned MASK = 0xffffffff03f          ;
```

```
*ptr = ( *ptr & MASK ) | ( val << (pos + 1 - len));
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = ~( 0x00000fc0 );
```

```
unsigned MASK = 0xfffff03f ;
```

```
*ptr = ( *ptr & MASK ) | ( val << (pos + 1 - len));
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = 0xffff03f;
```

```
*ptr = ( *ptr & MASK ) | ( val << (pos + 1 - len));  
date = ( date & 0xffff03f) | ( 7 << ( 11 + 1 - 6 ));
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = 0xfffff03f;
```

```
*ptr = ( *ptr & MASK ) | ( val << (pos + 1 - len));
```

```
date = ( date & 0xfffff03f ) | ( 7 << ( 11 + 1 - 6 ));
```

```
date = ( 27609578 & 0xfffff03f ) | ( 0x00000007 << ( 11 + 1 - 6 ));
```


Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = 0xffff03f;
```

```
date = ( date & 0xffff03f) | ( 7 << ( 11 + 1 - 6 ));
```

```
date = ( 27609578 & 0xffff03f) | (0x00000007 << ( 11 + 1 - 6 ));
```

```
date = (0x01a549ea & 0xffff03f) | (0x00000007 << 6 );
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = 0xffff03f;
```

```
date = ( 27609578 & 0xffff03f) | (0x00000007 << ( 11 + 1 - 6 ));
```

```
date = (0x01a549ea & 0xffff03f) | (0x00000007 << 6 );
```

```
date = 0x01a5402a | 0x000001c0 ;
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = 0xffff03f;
```

```
date = (0x01a549ea & 0xffff03f) | (0x00000007 << 6);
```

```
date = 0x01a5402a | 0x000001c0;
```

```
date = 0x01a541ea;
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = 0xfffff03f;
```

```
date = 0x01a5402a | 0x000001c0 ;
```

```
date = 0x01a541ea ;
```

```
date = 27607530 ;
```

Bitenkénti logikai műveletek

Példa [2/2]

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = 0xffff03f;
```

```
date = 0x01a541ea ;
```

```
date = 27607530 ;
```

- A függvények működése:

```
getBits(27609578, 16, 5);
```

```
return 20;
```

```
setBits(7, 11, 6, &date);
```

```
unsigned MASK = 0xffff03f;
```

```
date = 27607530;
```

C műveletek prioritása

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

Típusdefiníció C-ben

- A C nyelvben lehetőségünk van típusok tetszésünk szerinti elnevezésére, azaz típusdefinícióra melyet a `typedef` kulcsszó vezet be, alakja:

```
typedef típus új_típusnév;
```

- A definíciótól kezdve a `típus`-ra az `új_típusnév` azonosítóval (is) hivatkozhatunk.
- A típusdefinícióval élhetünk például akkor, mikor több helyen kell ugyanolyan típusú változót deklarálni, de ez a típus
 - a jövőbeni fejlesztések során esetleg változhat, vagy
 - egy bonyolult módon megadható típus, amit nehézkes lenne többször leírni (és a többszöri leírás melleleg hibaforrás is)!

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


Felsorolás adattípusok a C nyelvben

- Felsorolás adattípus értékalmaza a típusképzésben felsorolt azonosítók, mint konstans azonosítók által meghatározott értékek.
- Felsorolás adattípus esetén az értékadás műveletét és a relációs műveleteket szokás értelmezni, ahol a $<$ rendezési relációt a típusképzésben az elemek felsorolásával definiáljuk.
- Felsorolás adattípust az enum kulcsszóval definiálhatunk:

```
enum {elem1, ..., elemn}
```

- Például:

```
// típusdefiníció:  
typedef enum {hetfo, kedd, szerda, csutortok, pentek, szombat, vasarnap} het_t;  
// változódeklaráció előre definiált típussal:  
het_t nap;  
// változódeklaráció közvetlen típusmegadással:  
enum {hetfo, kedd, szerda, csutortok, pentek, szombat, vasarnap} nap;
```

Felsorolás adattípusok a C nyelvben

- A felsorolás típus C nyelven nagyon szorosan kötődik az `int` típushoz.
- Alapesetben a felsorolásban szereplő első azonosító értéke 0, a többié 1-gyel több az őt közvetlenül megelőzőnél. Ezzel egyben a rendezési relációt is definiáltuk.
- A C nyelv lehetőséget ad viszont a felsorolt azonosítók értékének közvetlen megadására.

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

- A típusképzésben felsorolt azonosítók úgy működnek, mintha abban a blokkban deklarált konstans azonosítók lennének, amelyek blokkban a típusdefiníció szerepel. Így a C nyelvben az is lehetséges, hogy egy típuson belül több azonosító ugyanazt az értéket kapja.

```
enum {a = 3, b, c, d = 2, e, f} ertek;
```

Felsorolás adattípusok a C nyelvben

- A kapcsolat a két típus között azért erős, mert a fordító az enum-ot teljes mértékben az int típusra vezeti vissza, ezért ennek a típusnak a műveletei **megfelelő körültekintéssel** használhatóak az enum típuson is.
 - Azért kell a megfelelő körültekintés, mert például a vasarnap + 1 érték már nem értelmezhető a het_t típus értékeként.

```
typedef enum {hetfo=1, keddi, szerda, csutortok, pentek, szombat, vasarnap} het_t;
het_t nap;
het_t KovNap(Het n) {
    if (n == vasarnap) {
        return hetfo;
    }
    return n + 1;
}

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

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

- Ha nem szeretnénk az `stdbool.h` standard típusát használni, akkor magunk is definiálhatunk logikai konstansokat:

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

vagy

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

- Esetleg bevezethetünk saját típust is:

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

Arra ügyelni kell, hogy továbbra sem csak a `true` vagy `TRUE` érték lesz logikai igaz értéként értelmezve!

Valós típusok a C nyelvben

Típusok és módosítók

- A C nyelvben a valós adattípusok a float és double.
- A double adattípus az alábbi kulcsszóval módosítható:
`long` Implementációfüggő módon 64, 80, 96 vagy 128 bites pontosságot megvalósító adattípus. A tárolási mérete különféle megfontolásokból a valós pontosság által igényeltnél nagyobb lehet (pl. egy 80 bites pontosságú érték 96 biten tárolódhat).
- A valós adattípusok az értékkészlet határain belül sem képesek minden valós értéket pontosan ábrázolni. Viszont az értékkészlet határain belüli minden a valós értéket képesek egy típusfüggő e relatív pontossággal ábrázolni, az a -hoz legközelebbi a típus által pontosan ábrázolható x valós értékkel. $(|(x - a)/a| \leq e)$
- Az egyes típusok méretére az alábbiak biztosan teljesülnek:
`sizeof(float) < sizeof(double) ≤ sizeof(long double)`

Valós típusok a C nyelvben

Műveletek

- A C nyelv különféle valós adattípusai az értékalmazukban különböznek egymástól, az értelmezett műveletükben megegyeznek.
- Valós kifejezésben bármely valós vagy egész típusú tényező (akár vegyesen többféle is) szerepelhet.
- Valós konstans típusa `double`, vagy a száMLEÍrásban megadott típus (`f`, `l` suffix).
- ÉrtéKadó művelet jobb oldalán álló kifejezés kiértékelése független attól, hogy a bal oldalon milyen típusú változó van.
- A típus pontatlansága miatt az `==` műveletet nagyon körültekintően kell használni!

Háromszög osztályozása, toleranciával [1/2]

haromszog-tol.c [1-28]

```
1 /* Milyen háromszöget határoz meg három pozitív valós szám,  
2  * mint a háromszög három oldalhosszúsága? A valós típus  
3  * pontatlansága miatt toleranciával számolunk.  
4  * 1997. Október 13. Dévényi Károly, devenyi@inf.u-szeged.hu  
5  * 2014. Február 19. Gergely Tamás, gertom@inf.u-szeged.hu  
6  */  
7  
8 #include <stdio.h>  
9  
10 #define EQUALS(X,Y) ( ( ((X) > (Y)) ? ((X) - (Y)) : ((Y) - (X)) ) <= 1e-10 )  
11  
12 int main() {  
13     double a, b, c;    /* a háromszög oldalhosszúságai */  
14     double m;          /* munkaváltozó a cseréhez */  
15  
16     printf("Kérem a három pozitív valós számot!\n");  
17     scanf("%lf%lf%lf", &a, &b, &c);  
18  
19     /* a,b,c átrendezése úgy, hogy a>=b,c legyen */  
20     if (a < b) {        /* a és b átrendezése */  
21         m = a;  
22         a = b;  
23         b = m;  
24     }  
25     if (a < c) {        /* a és c átrendezése */  
26         m = a;  
27         a = c;  
28         c = m;
```


Háromszög osztályozása, toleranciával [2/2]

haromszog-tol.c [30–49]

```
30                                     /* osztályozás */
31 if (c <= 0 || b <= 0) {
32     printf("Nem_háromszög!\n");
33 } else if (a >= b + c) {
34     printf("Nem_háromszög!\n");
35 } else if (EQUALS(a,b) && EQUALS(b,c) && EQUALS(a,c)) {
36     printf("Szabályos_háromszög.\n");
37 } else if (EQUALS(a,b) || EQUALS(b,c) || EQUALS(a,c)) {
38     if (EQUALS((a * a), (b * b + c * c))) {
39         printf("Egyenlőszárú_derékszögű_háromszög.\n");
40     } else {
41         printf("Egyenlőszárú_háromszög.\n");
42     }
43 } else if (EQUALS((a * a), (b * b + c * c))) {
44     printf("Derékszögű_háromszög.\n");
45 } else {
46     printf("Egyéb_háromszög.\n");
47 }
48                                     /* vége a többszörös szelekciónak */
49 return 0;
```

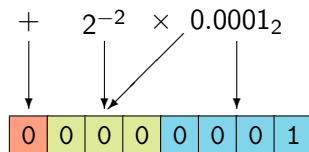
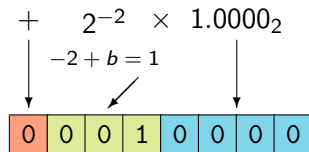
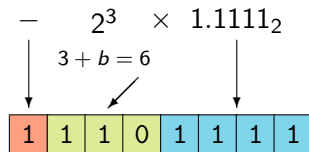
- Egy valós értéket tároló memóriaterület három részre osztható: az *előjelbitet*, a *törtet* és az exponenciális *kitevőt* kódoló részre.
- Az értékhalmoz szokásos leképezése a következő:
 - Az *előjelbit* 0 értéke a pozitív, 1 értéke a negatív számokat jelöli.
 - A számot kettes számrendszerben $1.m \times 2^k$ alakra hozzuk, majd az m számjegyeit eltároljuk a *törtnek*, a k -nak egy típusfüggő b konstanssal növelt értékét pedig a *kitevőnek* fenntartott részen.
 - Így a *tört* rész hossza az ábrázolás pontosságát (az értékes számjegyek számát), a *kitevő* pedig az értéktartomány méretét határozza meg.
 - Nagyon kicsi számokat speciálisan $0.m \times 2^{1-b}$ alakban tárolhatunk, ekkor a *kitevő* összes bitje 0.
 - Ha a *kitevő* összes bitje 1, az csupa 0 bitből álló *tört* esetén a ∞ , minden más esetben NaN értéket jelenti.
 - A 32/64 bites float/double az 1 *előjelbit* mögött 8/11 biten a *kitevő* $b = 127$ -tel/1023-mal növelt értékét, majd 23/52 biten a *törtet* tárolja.

Valós típusok a C nyelvben

Tárolás [2/3]

- Ha például 8 bites valós számokkal dolgoznánk, aminek 4 bitje a *tört* és 3 bitje a *kitevő* $b = 3$ korrekciós értékkel, akkor

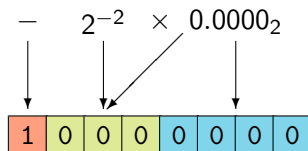
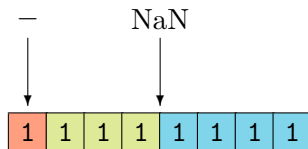
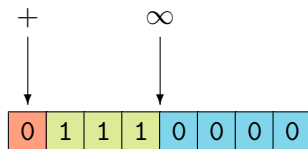
- a legkisebb ábrázolható érték a $-15.5 = -1.1111_2 \times 2^3$,
- a legkisebb „normálisan” ábrázolható pozitív érték az $1/4 = 1.0000_2 \times 2^{-2}$,
- a legkisebb ábrázolható pozitív érték pedig az $1/64 = 0.0001_2 \times 2^{-2}$ lenne.



Valós típusok a C nyelvben

Tárolás [3/3]

- Speciális értékeket is tárolhatunk:
 - A $\pm\infty$ értéket csupa 1-es kitevővel és csupa 0 tört alakban ($\pm 1.0000_2 \times 2^4$ -ként),
 - a $\pm\text{NaN}$ értéket csupa 1-es kitevővel és **nem** csupa 0 törttel (például $\pm 1.1111_2 \times 2^4$ -ként),
 - a ± 0 értéket pedig $\pm 0.0000_2 \times 2^{-2}$ alakban tárolhatjuk.



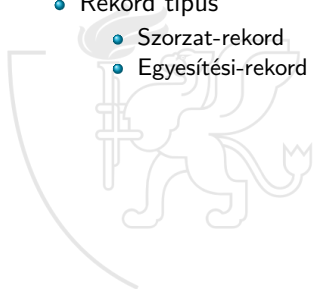
- 1 **Bemutakozás**
 - Kurzus információk
 - A SZTE és az informatikai képzés
- 2 **Linux**
 - Alapfogalmak
 - Linux parancsok
 - Linux shell
 - Felhasználók
 - Hálózat
- 3 **Gyors C áttekintés**
 - Bevezető
 - Pénzváltás (1. verzió)
 - Pénzváltás (2. verzió)
 - Röppálya számítás
 - Röppálya szimuláció
 - Az év napja
 - Csúszóátlag adott elemszámmra
 - Csúszóátlag parancssorból
 - Basename standard inputról
 - Basename parancssorból
 - Tér legtávolabbi pontjai
 - A nappalis gyakorlat értékelése

- 4 **Alapok**
 - Alapfogalmak
 - A programozás fázisai
 - Algoritmus vezérlése
 - A C nyelvű program
 - Szintaxis
 - A C nyelv elemi adattípusai
 - A C nyelv utasításai
- 5 **Vezérlési szerkezetek**
 - Bevezetés
 - Szekvenciális vezérlés
 - Függvények
 - Szelekciós vezérlések
 - Ismétléses vezérlések 1.
 - Eljárásvezérlés
 - Ismétléses vezérlések 2.
- 6 **Folyamatábra és struktúradiagram**
- 7 **Adatszerkezetek**
 - Az adatkezelés szintjei
 - Elemi adattípusok
 - **Pointer adattípus**
 - Tömb adattípus

- 8 **Sztringek**
 - Pointerek és tömbök C-ben
 - Rekord adattípus
 - Függvény pointer
 - Halmaz adattípus
 - Flexibilis tömbök
 - Láncolt listák
 - Típusokról C-ben
- 9 **IO**
 - Alapok
 - Adatállományok
- 10 **C fordítás**
 - A fordítás folyamata
 - A preprocesszor
 - A C fordító
 - Assembler
 - Linker és modulok
- 11 **Gyakorlati kérdések**
 - Memóriahasználát
 - Gyakori C hibák
 - where.c felboncolva

Összetett típusok, típusképzések

- Az összetett adattípusok értékei tovább bonthatóak, további értelmezésük lehetséges.
- A C nyelv összetett adattípusai:
 - Pointer típus
 - Függvény típus
 - Tömb típus
 - Sztringek
 - Rekord típus
 - Szorzat-rekord
 - Egyesítési-rekord



Összetett adattípusok, típusképzések

- 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, és
 - széles körben használható.



- 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.”



- Az eddigi tárgyalásunkban szerepelt változók statikusak abban az értelemben, hogy létezésük annak a blokknak a végrehajtásához kötött, amelyben a változó deklarálva lett. A programozónak a deklaráció helyén túl nincs befolyása a változó létesítésére és megszüntetésére.
 - (Ez a fajta statikusság nem tévesztendő össze azzal, amit a `static` kulcsszóval érhetünk el a C nyelvben!)
- 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 típusú változót a * segítségével deklarálhatunk:

```
típus * változónév;
```

- Például:

- char típusú dinamikus változó deklarálása:

```
char * pc;
```

- unsigned short int típusú dinamikus változó deklarációja:

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

- Meg kell jegyezni, hogy a * a változóhoz kötődik, vagyis

```
int * p;
```

kétféle értelmezése közül, miszerint

- A *p egy int típusú (dinamikus) változó, illetve
- A p egy int* típusú (int-re mutató) változó,

az első célravezetőbb a következő deklarációk értelmezésénél:

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

Ez ugyanis azt jelenti, hogy

- *p és q is int típusú változók, azaz csak p lesz pointer, és
 - hibás az az értelmezés, miszerint p és q is int* típusú pointer.
- Ezért szokás a típusképző *-ot szorosan a változóhoz, és nem a típushoz írni.

- Pointer típust az alábbi módon definiálhatunk:

```
typedef típus *pointertípusnév;
```

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

- Például:

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



- 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*



- A *változóhivatkozás* szintaktikus egység, meghatározott formai szabályok szerint képzett jelsorozat egy adott programnyelven, tehát egy kódrészlet.
- A *változó* a program futása során a program által lefoglalt memóriaterület egy része, amelyen egy adott (elemi vagy összetett) típusú *érték* tárolódik.
- 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.

- 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.



- **NULL**
 - Konstans, érvénytelen pointer érték, a pointerhez nem tartozik dinamikus változó.
- **Létesít**($\leftarrow X:PE$)
 - Új dinamikus változó létesítése.
- **Értékadás**($\leftarrow X:PE; \rightarrow Y:PE$)
 - Értékadás.
- **Törlés**($\leftrightarrow X:PE$)
 - Dinamikus változó törlése.
- **Dereferencia**($\rightarrow X:PE$)
 - A pointer által mutatott dinamikus változó elérése, eredménye egy változóhivatkozás.
- **Egyenlő**($\rightarrow p:PE; \rightarrow q:PE$): `bool`
 - Két pointer értéke megegyezik-e?
- **NemEgyenlő**($\rightarrow p:PE; \rightarrow q:PE$): `bool`
 - Két pointer értéke különbözik?

Pointer

Virtuális adattípus – Műveletek

- NULL

```
NULL
```

- Létesít($\leftarrow X:PE$)

```
X = malloc(sizeof(E));
```

- Értékadás($\leftarrow X:PE; \rightarrow Y:PE$)

```
X = Y;
```

- Törlés($\leftrightarrow X:PE$)

```
free(X);
```

- Dereferencia($\rightarrow X:PE$)

```
*X
```

- Egyenlő($\rightarrow p:PE; \rightarrow q:PE$): bool

```
p == q
```

- NemEgyenlő($\rightarrow p:PE; \rightarrow q:PE$): bool

```
p != q
```

- A C nyelv memóriaműveleteinek használatához szükség van az `stdlib.h` vagy a `memory.h` használatára.
- `malloc(S)`: lefoglal egy `S` méretű memóriaterületet a program számára.
- `sizeof(E)`: megadja, hogy egy `E` típusú változó mekkora helyet igényel a memóriában.
- `malloc(sizeof(E))`: hatására tehát létrejön egy új `E` típusú érték tárolására (is) alkalmas változó, és ez a változó lesz a `p` értéke.
- `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 típus műveletei

Példa

- A dinamikus változó létrehozására tehát a `malloc()`, megszüntetésére a `free()` függvény szolgál:

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

Változó	Érték
---------	-------

–	–
---	---

–	–
---	---

változóhivatkozás
a forráskódban

változó és értéke
a memóriában

p

*p

Pointer típus műveletei

Példa

- A dinamikus változó létrehozására tehát a `malloc()`, megszüntetésére a `free()` függvény szolgál:

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

Változó	Érték
---------	-------

p	-
---	---

-	-
---	---

változóhivatkozás
a forráskódban

változó és értéke
a memóriában

p



*p

Pointer típus műveletei

Példa

- A dinamikus változó létrehozására tehát a `malloc()`, megszüntetésére a `free()` függvény szolgál:

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

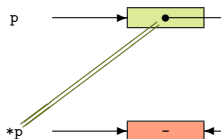
Változó	Érték
---------	-------

p	*p
---	----

*p	-
----	---

változóhivatkozás
a forráskódban

változó és értéke
a memóriában



Pointer típus műveletei

Példa

- A dinamikus változó létrehozására tehát a `malloc()`, megszüntetésére a `free()` függvény szolgál:

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

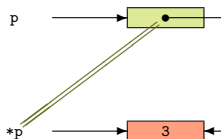
Változó	Érték
---------	-------

p	*p
---	----

*p	3
----	---

változóhivatkozás
a forráskódban

változó és értéke
a memóriában



Pointer típus műveletei

Példa

- A dinamikus változó létrehozására tehát a `malloc()`, megszüntetésére a `free()` függvény szolgál:

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

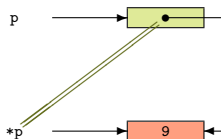
Változó	Érték
---------	-------

p	*p
---	----

*p	9
----	---

változóhivatkozás
a forráskódban

változó és értéke
a memóriában



Pointer típus műveletei

Példa

- A dinamikus változó létrehozására tehát a `malloc()`, megszüntetésére a `free()` függvény szolgál:

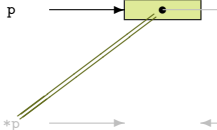
```
int *p;  
  
p = malloc(sizeof(int));  
*p = 3;  
  
*p += 6;  
free(p);
```

Változó	Érték
---------	-------

p	-
-	-

változóhivatkozás
a forráskódban

változó és értéke
a memóriában



Pointer típus műveletei

Példa

- A dinamikus változó létrehozására tehát a `malloc()`, megszüntetésére a `free()` függvény szolgál:

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

Változó	Érték
---------	-------

—	—
—	—
—	—

változóhivatkozás
a forráskódban

változó és értéke
a memóriában

p

q

*p

*q

Pointer típus műveletei

Példa

- A dinamikus változó létrehozására tehát a `malloc()`, megszüntetésére a `free()` függvény szolgál:

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

Változó	Érték
---------	-------

p	NULL
---	------

-	-
---	---

-	-
---	---

változóhivatkozás
a forráskódban

változó és értéke
a memóriában

p



q

*p

*q

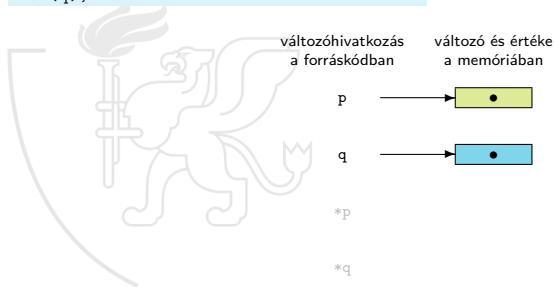
Pointer típus műveletei

Példa

- A dinamikus változó létrehozására tehát a `malloc()`, megszüntetésére a `free()` függvény szolgál:

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

Változó	Érték
p	NULL
q	NULL
–	–



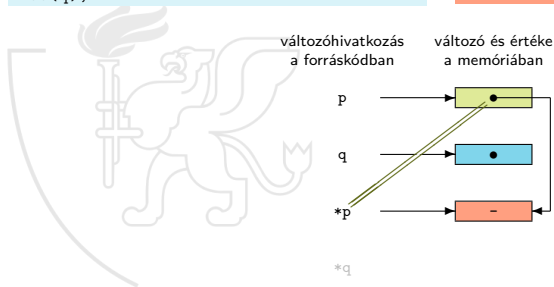
Pointer típus műveletei

Példa

- A dinamikus változó létrehozására tehát a `malloc()`, megszüntetésére a `free()` függvény szolgál:

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

Változó	Érték
p	din.v.
q	NULL
din.v.	-



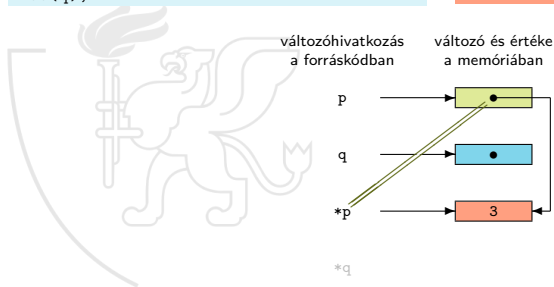
Pointer típus műveletei

Példa

- A dinamikus változó létrehozására tehát a `malloc()`, megszüntetésére a `free()` függvény szolgál:

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

Változó	Érték
p	din.v.
q	NULL
din.v.	3



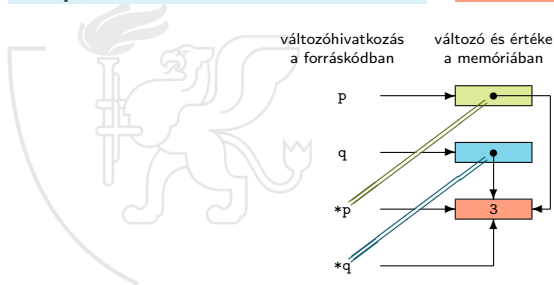
Pointer típus műveletei

Példa

- A dinamikus változó létrehozására tehát a `malloc()`, megszüntetésére a `free()` függvény szolgál:

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

Változó	Érték
p	din.v.
q	din.v.
din.v.	3



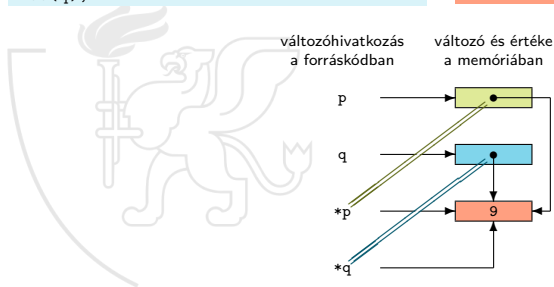
Pointer típus műveletei

Példa

- A dinamikus változó létrehozására tehát a `malloc()`, megszüntetésére a `free()` függvény szolgál:

```
int *p = NULL;  
int *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 műveletei

Példa

- A dinamikus változó létrehozására tehát a `malloc()`, megszüntetésére a `free()` függvény szolgál:

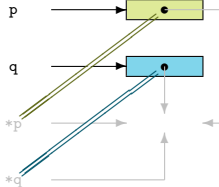
```
int *p = NULL;  
int *q = NULL;  
p = malloc(sizeof(int));  
*p = 3;  
q = p;  
*q += 6;  
free(q);
```

Változó	Érték
p	–
q	–
–	–



változóhivatkozás
a forráskódban

változó és értéke
a memóriában



Pointer

Virtuális adattípus – Memóriacím

- Linux alatt logikailag minden programnak saját memória-tartománya van, amin belül az egyes memóriacímeket egy sorszám azonosítja.
- Pointer típusú változó 32 bites rendszereken 4 bájt, 64 bites rendszereken 8 bájt hosszban a hozzá tartozó dinamikus változóhoz foglalt memóriamező kezdőcímét (sorszámát) tartalmazza.
- A pointer értéke tehát (második megközelítésben) értelmezhető egy tetszőleges memóriacímként is, amely értelmezés egybeesik a pointer megvalósításával.



- Ilyen módon viszont értelmezhetjük a címképző műveletet, ami egy változó memóriabeli pozícióját, címét adja vissza.

- $\text{Cím}(\rightarrow v:E; \leftarrow p:PE)$

```
p = &v;
```

- Az $\&$ művelet tehát egy változó memóriacímét adja vissza, így egy pointer értéke akár egy globális vagy lokális statikus változó is lehet.
- Ha nem statikus lokális változó címével dolgozunk, akkor arra azért figyeljünk, hogy ezek a változók csak addig léteznek, míg az adott blokkból ki nem lépünk.

Pointer típus műveletei

Példák

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

```
int i,j,*p;  
  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++; */
```



C műveletek prioritása

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

- 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 minden típusú pointerrel kompatibilisek értékadás és összehasonlítás tekintetében.



Függvény argumentumok

Kimenő mód kezelése

- A függvénytípusnál az argumentumok kezelése érték szerint történik, amely csak bemenő módú argumentumok használatát teszi lehetővé. Ha kimenő módú argumentumokra is szükségünk van, akkor ennek kezelését nekünk kell megoldani pointer segítségével.
- Legyen a csere függvény feladata az argumentumok értékének megcserélése. Ha a csere függvény deklarációja

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

akkor a

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

művelet nem végzi el a cserét, hiszen az csak az a és b változók értékét kapja meg és ezeket cserélgeti a lokális változóiban.

Függvény argumentumok

Kimenő mód kezelése

- Az a és b változók értéke helyett tehát a változók címét kell átadni és átvenni. Ha a csere függvény deklarációja

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

akkor a

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

művelet már megcseréli az a és b változók értékét.

Függvény argumentumok

- Ha tehát nem használunk globális változókat, akkor be- és kimenő módú argumentum kezelésére kielégítő megoldást ad a következő séma:
 - A paraméterek értékét lokális változóba tesszük, majd ezek értékeit a `return` utasítás előtt visszaadjuk a paraméternek. (Tisztán kimenő módú paraméter esetén a paraméter értékének lokális változóba másolása (◀) elhagyható.)
 - A hívás helyén nem a változó értékét, hanem a címét adjuk át argumentumként.

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


Másodfokú egyenlet gyökei

Problémafelvetés és specifikáció

- Problémafelvetés
 - Határozzuk meg egy másodfokú egyenlet valós gyökeit!
- Specifikáció
 - A probléma inputja három valós szám, az $ax^2 + bx + c$ másodfokú egyenlet a , b és c együtthatói.
 - Az output két valós szám, az egyenlet x_1 , x_2 valós gyökei, ha létezik valós gyök, különben – ha nincs valós gyök, vagy végtelen sok van – egy szöveg, ami a valós gyökök kiszámíthatatlanságát jelzi.



Másodfokú egyenlet gyökei

Algoritmustervezés

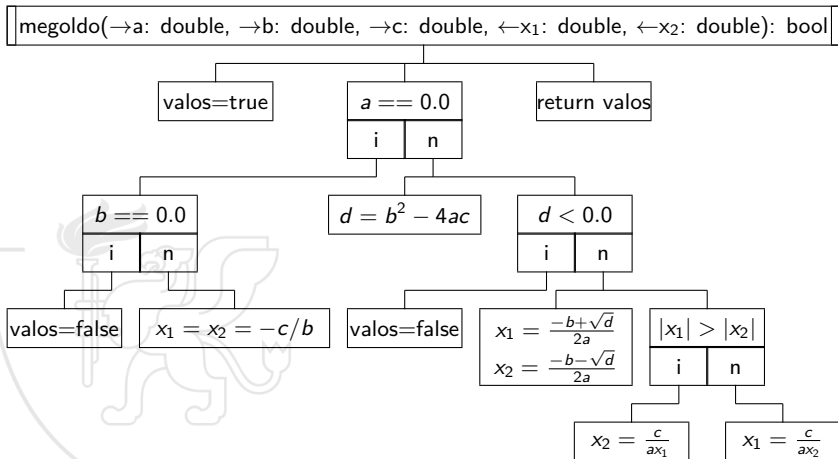
- A probléma megoldása olyan függvényművelettel adható meg, amelynek
 - három bemenő paramétere van: a , b , c , az egyenlet együtthatói;
 - két kimenő paramétere van: x_1 , x_2 , a két valós gyök;
 - továbbá a függvény logikai értéket ad vissza, amely akkor és csak akkor lesz igaz, ha van valós gyök.
- A közismert $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ megoldóképletet használjuk.



Másodfokú egyenlet gyökei

Algoritmustervezés – Szerkezeti ábra

- A függvény:



Másodfokú egyenlet gyökei [1/3]

masodfok.c [1-9]

```
1 /* Másodfokú egyenlet valós gyökeinek meghatározása a  
2  * megoldó függvényekkel.  
3  * 1998. március 31. Dévényi Károly, devenyi@inf.u-szeged.hu  
4  * 2013. augusztus 29. Gergely Tamás, gertom@inf.u-szeged.hu  
5  */  
6  
7 #include <stdio.h>  
8 #include <math.h>  
9 #include <stdbool.h>
```



Másodfokú egyenlet gyökei [2/3]

masodfok.c [11–37]

```
11 bool megoldo(double a, double b, double c, double *x1, double *x2) {
12     bool valos = true;                                /* van-e megoldás */
13
14     if (a == 0.0) {
15         if (b == 0.0) {                                /* az egyenlet elfajuló */
16             valos = false;
17         } else {
18             *x1 = *x2 = -(c / b);
19         }
20     } else {
21         double d;                                       /* a diszkrimináns */
22         d = b * b - 4.0 * a * c;
23         if (d < 0.0) {                                   /* nincs valós gyöke */
24             valos = false;
25         } else {
26             *x1 = (-b + sqrt(d)) / (2.0 * a);
27             *x2 = (-b - sqrt(d)) / (2.0 * a);
28             if (fabs(*x1) > fabs(*x2)) { /* gyökök pontosabb kiszámolása */
29                 *x2 = c / (*x1 * a);
30             } else if (*x2 != 0) {
31                 *x1 = c / (*x2 * a);
32             }
33         }
34     }
35
36     return valos;
37 }
```

Másodfokú egyenlet gyökei [3/3]

masodfok.c [39–57]

```
39 int main() {
40     double a, b, c, x, y;                                /* a főprogram változói */
41
42     printf("Kérem az első egyenlet együtthatóit!\n");
43     scanf("%lg%lg%lg%*[^\\n]", &a, &b, &c); getchar();
44     if (megoldo(a, b, c, &x, &y)) {
45         printf("Az egyenlet gyökei: %20.10f %20.10f\\n", x, y);
46     } else {
47         printf("Az egyenletnek nincs valószínű megoldása!\\n");
48     }
49     printf("Kérem a második egyenlet együtthatóit!\n");
50     scanf("%lg%lg%lg%*[^\\n]", &a, &b, &c); getchar();
51     if (megoldo(a, b, c, &x, &y)) {
52         printf("Az egyenlet gyökei: %20.10f %20.10f\\n", x, y);
53     } else {
54         printf("Az egyenletnek nincs valószínű megoldása!\\n");
55     }
56     return 0;
57 }
```



Másodfokú egyenlet gyökei

- Az x_1 és x_2 kimenő paraméterek, ezért átalakítjuk a függvény deklarációját. Mivel nem használunk globális változókat, ezek nem okozhatnak problémát.



Másodfokú egyenlet gyökei [megoldo függvény változat]

masodfok.c [11–37] helyett

```
11 bool megoldo(double a, double b, double c, double *x1, double *x2) {
12     bool valos = true;                                /* van-e megoldás */
13     double mx1, mx2;                                  /* munkaváltozók *x1 és *x2 helyett */
14     if (a == 0.0) {
15         if (b == 0.0) {                                /* az egyenlet elfajuló */
16             valos = false;
17         } else {
18             mx1 = mx2 = -(c / b);
19         }
20     } else {
21         double d;                                      /* a diszkrimináns */
22         d = b * b - 4.0 * a * c;
23         if (d < 0.0) {                                  /* nincs valós gyöke */
24             valos = false;
25         } else {
26             mx1 = (-b + sqrt(d)) / (2.0 * a);
27             mx2 = (-b - sqrt(d)) / (2.0 * a);
28             if (fabs(mx1) > fabs(mx2)) { /* gyökök pontosabb kiszámolása */
29                 mx2 = c / (mx1 * a);
30             } else if (mx2 != 0) {
31                 mx1 = c / (mx2 * a);
32             }
33         }
34     }
35     *x1 = mx1; *x2 = mx2;
36     return valos;
37 }
```


- 1 **Bemutakozás**
 - Kurzus információk
 - A SZTE és az informatikai képzés
- 2 **Linux**
 - Alapfogalmak
 - Linux parancsok
 - Linux shell
 - Felhasználók
 - Hálózat
- 3 **Gyors C áttekintés**
 - Bevezető
 - Pénzváltás (1. verzió)
 - Pénzváltás (2. verzió)
 - Röppálya számítás
 - Röppálya szimuláció
 - Az év napja
 - Csúszóátlag adott elemszámmra
 - Csúszóátlag parancssorból
 - Basename standard inputról
 - Basename parancssorból
 - Tér legtávolabbi pontjai
 - A nappalis gyakorlat értékelése

- 4 **Alapok**
 - Alapfogalmak
 - A programozás fázisai
 - Algoritmus vezérlése
 - A C nyelvű program
 - Szintaxis
 - A C nyelv elemi adattípusai
 - A C nyelv utasításai
- 5 **Vezérlési szerkezetek**
 - Bevezetés
 - Szekvenciális vezérlés
 - Függvények
 - Szelekciós vezérlések
 - Ismétléses vezérlések 1.
 - Eljárásvezérlés
 - Ismétléses vezérlések 2.
- 6 **Folyamatábra és struktúradiagram**
- 7 **Adatszerkezetek**
 - Az adatkezelés szintjei
 - Elemi adattípusok
 - Pointer adattípus
 - **Tömb adattípus**

- 8 **Sztringek**
 - Pointerek és tömbök C-ben
 - Rekord adattípus
 - Függvény pointer
 - Halmaz adattípus
 - Flexibilis tömbök
 - Láncolt listák
 - Típusokról C-ben
- 9 **IO**
 - Alapok
 - Adatállományok
- 10 **C fordítás**
 - A fordítás folyamata
 - A preprocesszor
 - A C fordító
 - Assembler
 - Linker és modulok
- 11 **Gyakorlati kérdések**
 - Memóriahasználat
 - Gyakori C hibák
 - where.c felboncolva

- 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.
- Ekkor tehát egy olyan összetett adathalmazzal van dolgunk, amelynek egy eleme $A = (a_0, \dots, a_{n-1})$, ahol $a_i \in E, \forall i \in (0, \dots, n-1)$ -re.
- Ha az ilyen sorozatokon a következő műveleteket értelmezzük, akkor egy (absztrakt) adattípushoz jutunk, amit *Tömb* típusnak nevezünk.
- Jelöljük ezt a *Tömb* típust T -vel, a $0, \dots, n-1$ intervallumot pedig I -vel.

- $\text{Kiolvas}(\rightarrow A:T; \rightarrow i:I; \leftarrow x:E)$
 - Adott $i \in I$ -re az A sorozat i . komponensének kiolvasása adott x , E típusú változóba.
- $\text{Módosít}(\leftrightarrow A:T; \rightarrow i:I; \rightarrow x:E)$
 - Adott $i \in I$ -re az A sorozat i . komponensének módosítása adott x , E típusú értékre.
- $\text{Értékadás}(\leftarrow A:T; \rightarrow X:T)$
 - Értékadó művelet. Az A változó felveszi az X , T típusú kifejezés értékét.

- Problémafelvetés

- Adott valahány ember, akik riadóláncot akarnak alkotni. A közösség minden tagjára meghatározott, hogy kit értesít. Eldöntendő, hogy adott értesítési hozzárendelések valóban riadóláncot alkotnak-e?

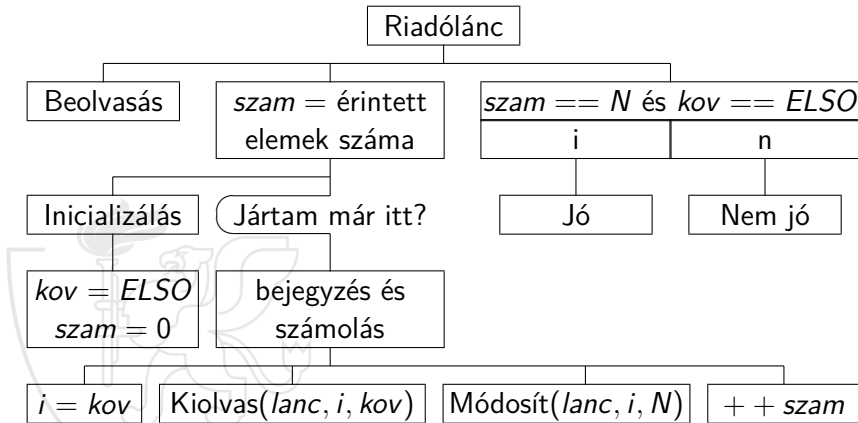
- Specifikáció

- A probléma inputja egy n szám, a közösség tagjainak (akiket $[0, \dots, n - 1]$ sorszámmal azonosítunk) száma, valamint egy n elemű számsorozat, melynek i . tagja az i . sorszámu ember által értesítendő személy sorszáma.
- Az output egy „Igen” vagy „Nem” tetszőleges szövegkörnyezetben.

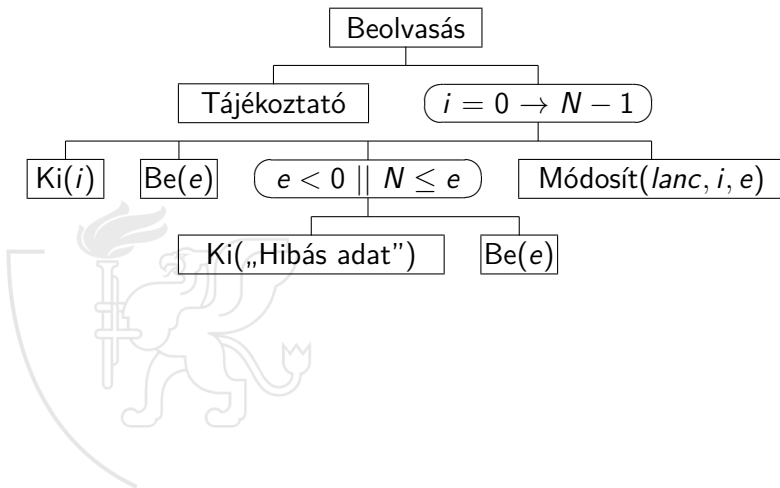
- A riadólánc egy *Lanc* nevű számsorozattal adható meg, amelynek i . eleme annak az embernek a sorszáma, akit az i -nek értesíteni kell.
- A *Lanc* sorozat akkor és csak akkor valódi riadólánc, ha bármelyik elemétől elindulva a *Lanc* szerinti hozzárendelést követve visszajutunk i -be úgy, hogy közben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



- Az program szerkezeti ábrája:



- A beolvasás szerkezeti ábrája:



- Tömb típusú változót az alábbi módon deklarálhatunk:

```
típus változónév[elemszám];
```

- Például:

```
char ct[5];           /* 5 char típusú elemből álló tömb */  
unsigned short int it[20]; /* 20 elemű unsigned short int tömb */
```

- 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;
```


- A Kiolvas és a Módosít műveletek megvalósítása a tömbelem-hivatkozással történik.
- A tömbelem-hivatkozásra a `[]` operátort használjuk. Ez egy olyan tömbökön értelmezett művelet C-ben, ami nagyon magas precedenciával rendelkezik és balasszociatív.
- Egy tömbre a tömbindexelés operátort (megfelelő index használatával) alkalmazva a tömb adott elemét változóként kapjuk vissza.



C műveletek prioritása

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

- $\text{Kiolvas}(\rightarrow A:T; \rightarrow i:I; \leftarrow x:E)$

```
x = A[i];
```

- $\text{Módosít}(\leftrightarrow A:T; \rightarrow i:I; \rightarrow x:E)$

```
A[i] = x;
```

- $\text{Értékadás}(\leftarrow A:T; \rightarrow X:T)$

- C-ben az $A = X$ művelet nem alkalmazható tömbökre, mert a baloldalon nem változóhivatkozás áll (erről később). (Az értékadás művelet nem keverendő össze az inicializálással!)
- Használható viszont az alacsony szintű `memcpy()`, vagy `C11`-től a biztonságosabb `memcpy_s()` függvény.

```
memcpy(A, X, sizeof(T));  
memcpy_s(A, sizeof(A), X, sizeof(T));
```

- C nyelvben a tömb indexelése minden esetben 0-val kezdődik, azaz egy

```
int t[20];
```

deklaráció esetén a tömb elemei $t[0]$, $t[1]$, ..., $t[19]$ lesznek.

- Nincs viszont indexhatár-ellenőrzés, azaz a fenti deklaráció esetén például a $t[20]$ vagy $t[-1]$ elemekre is lehet hivatkozni, ez azonban nagyon csúnya futási hibákat eredményezhet.



Riadólánc [1/2]

riadolanc.c [1–18]

```
1 /* Adott n számú embert tartalmazó közösség. Eldöntendő,  
2  * hogy riadóláncot alkotnak-e?  
3  * 2006. augusztus 9. Gergely Tamás, gertom@inf.u-szeged.hu  
4  */  
5  
6 #include <stdio.h>  
7  
8 #define N      8                                /* a sorozat elemeinek száma */  
9 #define ELSON  0                                /* az első vizsgálandó elem sorszáma */  
10  
11 int main () {  
12     int lanc[N];                                /* a sorozatot tároló tömb */  
13     int e, i;                                    /* munkaváltozók */  
14     int kov;                                      /* a következő vizsgálandó */  
15     int szam;                                     /* számláló */  
16  
17     printf("Kérem a %d elemű sorozatot, amelyről", N);  
18     printf("eldöntöm, hogy riadólánc-e!\n");
```

Riadólánc [2/2]

riadolanc.c [20–46]

```
20     for (i = 0; i < N; ++i) {                                     /* beolvasás */
21         printf("%d. kitértesít?", i);
22         scanf("%d*[^\\n]", &e); getchar();
23
24         while ((e < 0) || (N <= e)) {
25             printf("Hibás adat!\\nKérem újra:");
26             scanf("%d*[^\\n]", &e); getchar();
27         }
28         lanc[i] = e;
29     }
30
31     kov = ELSO;                                                  /* inicializálás */
32     szam = 0;
33     do {
34         i = kov;
35         kov = lanc[i];
36         lanc[i] = N;                                             /* továbblépés */
37         ++szam;                                                  /* jártam már itt bejegyzése */
38     } while (lanc[kov] != N);                                     /* jártam már itt? */
39
40     if ((szam == N) && (kov == ELSO)) {
41         printf("A számsorozat riadólánc.\\n");
42     } else {
43         printf("A számsorozat nem riadólánc.\\n");
44     }
45     return 0;
46 }
```

Általános tömb

Absztrakt típus – Értékkészlet

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



Általános tömb

Absztrakt adattípus – Műveletek

- $\text{Kiolvas}(\rightarrow A:T; \rightarrow i_1:I_1; \dots; \rightarrow i_k:I_k; \leftarrow x:E)$
 - Az A tömb (i_1, \dots, i_k) indexű komponensének kiolvasása adott x , E típusú változóba.
- $\text{Módosít}(\leftrightarrow A:T; \rightarrow i_1:I_1; \dots; \rightarrow i_k:I_k; \rightarrow x:E)$
 - Az A tömb (i_1, \dots, i_k) indexű komponensének módosítása adott x , E típusú értékre.
- $\text{Értékadás}(\leftarrow A:T; \rightarrow X:T)$
 - Értékadó művelet. Az A változó felveszi az X , T típusú kifejezés értékét.

- Vegyük észre, hogy a $T = \text{Tömb}(I_1, \dots, I_k, E)$ típus felfogható úgy, mint egy $T = \text{Tömb}(I_1, T_1)$, ahol $T_1 = \text{Tömb}(I_2, \dots, I_k, E)$.
- Továbbá az sem jelent megkötést, hogy minden egyes I_j intervallum $[0..N_j]$ alakú legyen, hiszen bármely $[n..m]$ intervallum elemei egyszerűen transzformálhatók a $[0..(m - n)]$ intervallumra.
- Ezek alapján a C nyelven már létre tudunk hozni többdimenziós tömb típust is:

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

- 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 egy `double` változó;
 - `m[5]` egy tömb, aminek egy eleme egy `double` változó;
 - `m[5][7]` egy `double` változó.

Általános tömb

Virtuális C adattípus – Műveletek

- $\text{Kiolvas}(\rightarrow A:T; \rightarrow i_1:I_1; \dots; \rightarrow i_k:I_k; \leftarrow x:E)$

```
x = A[i1]...[ik];
```

- $\text{Módosít}(\leftrightarrow A:T; \rightarrow i_1:I_1; \dots; \rightarrow i_k:I_k; \rightarrow x:E)$

```
A[i1]...[ik] = x;
```

- $\text{Értékadás}(\leftarrow A:T; \rightarrow X:T)$

- C-ben az $A = X$ művelet nem alkalmazható tömbökre, mert a baloldalon nem változóhivatkozás áll (erről később). (Az értékadás művelet nem keverendő össze az inicializálással!)
- Használható viszont az alacsony szintű `memcpy()`, vagy `C11`-től a biztonságosabb `memcpy_s()` függvény.

```
memcpy(A, X, sizeof(T));  
memcpy_s(A, sizeof(A), X, sizeof(T));
```

- Ha tömböt kezdőértékkel deklarálunk, akkor az *első* dimenzió mérete elhagyható, mert az a kezdőértékek számából kiolvasható. De ha kevesebb kezdőértéket adunk meg, mint amekkora tömbre később szükségünk van, akkor ezt a méretet is meg kell adni.

- Egy 3×3 -as tömb:

```
int t[][3] = {  
    { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }  
};
```

- Egy 4×3 -as tömb, amelyben az utolsó sort nem inicializáltuk:

```
int t[4][3] = {  
    { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }  
};
```

- Egy 4×3 -as tömb, amelyben csak az első oszlopot inicializáljuk:

```
int t[][3] = {  
    { 1 }, { 4 }, { 7 }, { 9 }  
};
```

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

- Ahhoz, hogy a tömb hatékony legyen olyan TCF függvényt keresünk, amely egyszerűen, gyorsan kiszámítható i_1, \dots, i_k függvényében.
- Egydimenziós tömb esetén adja magát a $i_1 c_1$ képlet, ahol c_1 egy tömbelem mérete.
- A képletet könnyen kiterjeszthetjük több dimenzióra:
 - Legyen a tömb definíciója $E \ T[N_1] \dots [N_k]$;
 - $T[i_1]$ offsetje (címe a tömbön belül) $i_1 \cdot \text{sizeof}(E[N_2] \dots [N_k])$
 - $T[i_1][i_2]$ offsetje $i_1 c_1 + i_2 \cdot \text{sizeof}(E[N_3] \dots [N_k])$
 - ...
 - $T[i_1] \dots [i_k]$ offsetje $i_1 c_1 + \dots + i_k c_k$, ahol
 $\forall 1 \leq i < k : c_i = \text{sizeof}(E[N_{i+1}] \dots [N_k])$ és $c_k = \text{sizeof}(E)$
- Ha a tömbméretet konstansként adtuk meg, akkor a c_1, \dots, c_k konstansok fordítási időben kiszámíthatóak.
- Ha a tömbünk méretét változóval adtuk meg, akkor a c_1, \dots, c_k konstansok kiszámítása futásidőben történik.

Tömb

Fizikai adattípus

- Ez tulajdonképpen a tömbelemek index szerinti lexikografikus rendezését adja meg: $(i_1, \dots, i_k) <_{lex} (j_1, \dots, j_k)$ akkor és csak akkor, ha a legkisebb u indexre, amelyre $i_u \neq j_u$, teljesül az $i_u < j_u$.
- Kétdimenziós esetben (E T[N][M] ;) például a lexikografikus rendezés az alábbiak szerint működik:

0x00000000 ↑	...	0	1		j		M-2	M-1
T[0][0]	0	0	1	...	j	...	M-2	M-1
T[0][1]	1	M	M+1	...	Mj	...	2M-2	2M-1
...								
T[0][M-1]	i
T[1][0]		iM	iM+1	...	iM+j	...	(i+1)M-2	(i+1)M-1
...								
T[N-2][M-1]	N-2
T[N-1][0]		(N-2)M	(N-2)M+1	...	(N-2)M+j	...	(N-1)M-2	(N-1)M-1
...								
T[N-1][M-1]	N-1	(N-1)M	(N-1)M+1	...	(N-1)M+j	...	NM-2	NM-1
0xffffffff ↓	...							

- A tömbök használata nagy körültekintést igényel, mert a program végrehajtása közben nincs indexhatár-ellenőrzés, így `double m[10][10];` deklaráció esetén az `m[0][10]` hivatkozás ugyanaz mint az `m[1][0]`, és a C mindkettőt elfogadja.
 - Ez persze kihasználható többdimenziós tömbök egyszerűbb inicializálásakor, vagy pozíciófüggetlen műveletek elvégzésekor.
 - Például itt az alábbi kódrészlet és a kimenete:

```
1 int i, j, a[3][3];
2 for(i=0; i<9; ++i)
3     a[0][i]=i;
4 for(i=0; i<3; ++i) {
5     for(j=0; j<3; ++j)
6         printf("%d ", a[i][j]);
7     printf("\n");
8 }
```

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

• Problémafelvetés

- 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.
- Készítsünk tehát egy olyan függvényt, amely egy értesítési láncról eldönti, hogy az riadólánc-e (matematikai megfogalmazással: eldönti, hogy a kapott adatok ciklikus permutációt határoznak-e meg) anélkül, hogy a láncot magát megváltoztatná.

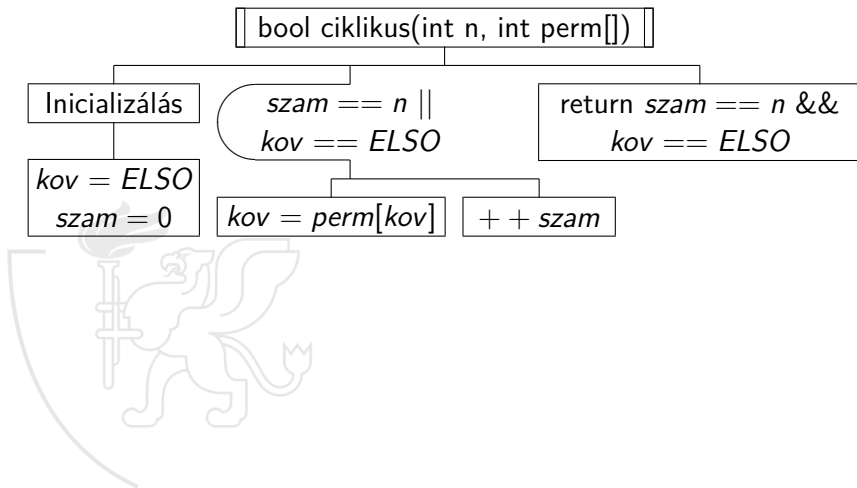
• Specifikáció

- A függvény inputja egy n szám, a közösség tagjainak (akiket $[0, \dots, n-1]$ sorszámokkal azonosítunk) száma, valamint egy n elemű tömb, melynek i . tagja az i . sorszámú ember által értesítendő személy sorszáma. A tömb bemenő paraméter, értékét a függvény nem változtathatja meg.
- Az output a függvény visszatérési értéke, egy logikai igaz érték ha az input ciklikus permutáció, illetve hamis ha nem az.

Ciklikus függvény

Algoritmustervezés – Szerkezeti ábra

- A ciklikus függvény szerkezeti ábrája:



Tömb mint paraméter

- Egy függvény paramétere lehet tömb típusú is. Ilyen esetben azonban nem történik teljes értékmásolás, hanem csak a tömb címe (vagyis egy pointer) kerül átadásra, így a tömb elemein végzett bármely módosítás az eredeti tömbön kerül végrehajtásra.
- 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.
- 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:

```
bool ciklikus(int n, int A[]);  
int fgv(int tomb[][10][3]);
```

Ciklikus függvény [1/2]

riadolanc-ciklikus.c [1-21]

```
1 /* Adott n számú embert tartalmazó közösség. Eldöntendő,  
2  * hogy riadoláncot alkotnak-e?  
3  * Ehhez felhasználunk egy függvényt, amely eldönti, hogy  
4  * a kapott sorozat ciklikus permutáció-e?  
5  * 2005. október 13. Gergely Tamás, gertom@inf.u-szeged.hu  
6  * 2014. október 9. Gergely Tamás, gertom@inf.u-szeged.hu  
7  */  
8  
9 #include <stdio.h>  
10 #include <stdbool.h>  
11  
12 #define N      8                                /* a sorozat elemeinek száma */  
13 #define ELSO 0                                /* az első vizsgálandó elem sorszáma */  
14  
15 bool ciklikus(int n, int perm[]) {              /* A paraméter egy tömb */  
16     int kov = ELSO, szam = 0;                  /* következő és számláló */  
17     do {  
18         kov = perm[kov]; szam++;                /* továbblépés */  
19     } while ((kov != ELSO) && (szam != n));  
20     return (kov == ELSO) && (szam == n);  
21 }
```

Ciklikus függvény [2/2]

riadolanc-ciklikus.c [23–47]

```
23 int main () {
24     int lanc[N];
25     int e, i;
26
27     printf("Kérem a %d elemű sorozatot, amelyről", N);
28     printf("eldöntöm, hogy riadólánc-e!\n");
29
30     for (i = 0; i < N; ++i) {
31         printf("%d. kit értesít?", i);
32         scanf("%d*[^\\n]", &e); getchar();
33
34         while ((e < 0) || (N <= e)) {
35             printf("Hibás adat!\nKérem újra:");
36             scanf("%d*[^\\n]", &e); getchar();
37         }
38         lanc[i] = e;
39     }
40
41     if (ciklikus(N, lanc)) {
42         printf("A számsorozat riadólánc.\n");
43     } else {
44         printf("A számsorozat nem riadólánc.\n");
45     }
46     return 0;
47 }
```

Tömbök

Egy komplexebb adatszerkezet megvalósítása

- 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?

Szomszédsági mátrix

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

Éllista

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

- 1 **Bemutakozás**
 - Kurzus információk
 - A SZTE és az informatikai képzés
- 2 **Linux**
 - Alapfogalmak
 - Linux parancsok
 - Linux shell
 - Felhasználók
 - Hálózat
- 3 **Gyors C áttekintés**
 - Bevezető
 - Pénzváltás (1. verzió)
 - Pénzváltás (2. verzió)
 - Röppálya számítás
 - Röppálya szimuláció
 - Az év napja
 - Csúszóátlag adott elemszámmra
 - Csúszóátlag parancssorból
 - Basename standard inputról
 - Basename parancssorból
 - Tér legtávolabbi pontjai
 - A nappalis gyakorlat értékelése

- 4 **Alapok**
 - Alapfogalmak
 - A programozás fázisai
 - Algoritmus vezérlése
 - A C nyelvű program
 - Szintaxis
 - A C nyelv elemi adattípusai
 - A C nyelv utasításai
- 5 **Vezérlési szerkezetek**
 - Bevezetés
 - Szekvenciális vezérlés
 - Függvények
 - Szelekciós vezérlések
 - Ismétléses vezérlések 1.
 - Eljárásvezérlés
 - Ismétléses vezérlések 2.
- 6 **Folyamatábra és struktúradiagram**
- 7 **Adatszerkezetek**
 - Az adatkezelés szintjei
 - Elemi adattípusok
 - Pointer adattípus
 - Tömb adattípus

- **Sztringek**
 - Pointerek és tömbök C-ben
 - Rekord adattípus
 - Függvény pointer
 - Halmaz adattípus
 - Flexibilis tömbök
 - Láncolt listák
 - Típusokról C-ben
- 8 **IO**
 - Alapok
 - Adatállományok
- 9 **C fordítás**
 - A fordítás folyamata
 - A preprocesszor
 - A C fordító
 - Assembler
 - Linker és modulok
- 10 **Gyakorlati kérdések**
 - Memóriahasználat
 - Gyakori C hibák
 - where.c felboncolva

A sztring adattípus [1/2]

- A program futása során a felhasználóval való kommunikáció általában a program által kiírt illetve a felhasználó által bevitt szövegeken, karaktersorozatokon keresztül történik.
- Ilyen karaktersorozatok tárolására kézenfekvő megoldást jelent a karakter tömb típus (rögzített E = karakter alaptípussal és $I = N^+$ indextípussal), ekkor a sztring adattípus értékkészlete a $0, 1, \dots, \infty$ hosszúságú karaktersorozatok halmaza.
- Ezen az értékhalmon értelmezett alapvetőbb sztring műveletek:
 - Karakter kiolvasása
 - Karakter módosítása
 - Hossz meghatározása
 - Értékadás
 - Összefűzés
 - Egyenlő

A sztring adattípus [2/2]

- Mivel azonban a végtelen az algoritmizálás során nem túl szerencsés fogalom, és a tömböknek is véges méretük van, a tényleges megvalósítás mindig valamilyen korlátos sztringet valósít meg.
- Így általában $\text{Sztring}(n)$ típusú sztringekről beszélhetünk, amiket $E =$ karakter alaptípusból és $I = 0, \dots, n - 1$ indextípusból képzett tömbökkel reprezentálhatunk, és amiknek az értékkészlete a legfeljebb n hosszúságú karaktersorozatok halmaza.
- Az ilyen módon különböző n méretekkel implementált sztringek viszont nagymértékben kompatibilisek egymással, műveleteik megegyeznek, és az értéktartományokon belül azonos eredményt adnak.

Sztringek C-ben [1/2]

- 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 méretére (a fizikai korlátokon kívül) nincs korlátozás.
- Az `str` szöveg i . karakterére az `str[i-1]` változóhivatkozással hivatkozhatunk.
- A sztring értékeket idézőjelek között lehet megadni.

```
"Helló_Világ!"  
"Hány_forintot_vált_sunk?"  
"%lf"  
"%lf_HUF_=_%lf_EUR\n"
```

- A karaktertömb számára lefoglalt hely (a sztring *mérete*) valójában csak egy felső korlátot jelent a sztring *hosszára* nézve. A sztring aktuális értéke ettől a korláttól lefelé bármikor eltérhet.
- A szöveg végét a szöveghez tartozó utolsó karakter után elhelyezett `'\0'` karakter jelzi. Így egy h méretű karaktertömbben maximum $h - 1$ értékes karaktert, ezáltal maximum $h - 1$ karakter hosszú szöveget tárolhatunk.
- Legyen `char str[6];` a sztring deklarációja, ekkor az "egy", "alma" és "meggy" szavak, valamint az üres sztring (""), az alábbi módon tárolódnak el.

0.	1.	2.	3.	4.	5.
e	g	y	\0		
a	l	m	a	\0	
m	e	g	g	y	\0
\0					

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

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

- Van azonban egy egyszerűbb forma is:

```
"string"
```

- A sztring literálban a legtöbb, a karakter literáloknál már megismert escape szekvencia használható. Az egyetlen különbség a határolójelekből adódik:
 - míg karakter esetén az aposztrófot kell escape szekvenciával megadni ('\'', '\"'),
 - addig sztringek esetében az idézőjelet ("'", "\"").
- Egymástól csak whitespace karakterekkel elválasztott sztring literálokat a fordító összefűzi, és egyetlen értékként kezeli.

Sztringek C-ben

Inicializálás

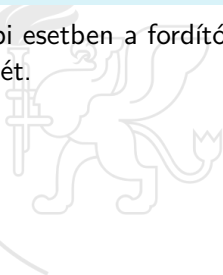
- Az ilyen módokon megadott sztringliterálok használhatók inicializálásra, azaz:

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

vagy akár

```
char str[] = "sztr" "ing";
```

Utóbbi esetben a fordító automatikusan határozza meg a sztring méretét.



Sztring műveletek C-ben.

Karakterek elérése

- **Kiolvas**($\rightarrow s$: Sztring, $\rightarrow i$: N^+ , $\leftarrow c$: Karakter)
 - Mivel a sztringek valójában tömbök, a tömbindexelés operátort (`[]`) használjuk.

```
c = s[i];
```

- **Módosít**($\leftrightarrow s$: Sztring, $\rightarrow i$: N^+ , $\rightarrow c$: Karakter)
 - Mivel a sztringek valójában tömbök, a tömbindexelés operátort (`[]`) használjuk.

```
s[i] = c;
```

Sztring műveletek C-ben.

Sztring hossza

- A sztringek hosszának meghatározása gyakran szükséges. Az `strlen()` függvénynek nagyon sokféle lehetséges implementációja lehetséges:

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

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

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

```
int strlen(char s[]) {  
    int i;  
    for (i = 0; s[i] != '\0'; ++i);  
    return i;  
}
```

```
int strlen(char *s) {  
    int n;  
    for (n = 0; *s != '\0'; ++s, ++n);  
    return n;  
}
```

```
int strlen(char *s) {  
    char *p = s;  
    while (*p) ++p;  
    return p - s;  
}
```

```
int strlen(char *s) {  
    char *p;  
    for (p = s; *p; ++p);  
    return p - s;  
}
```


Sztring műveletek C-ben.

string.h

- C nyelvben a sztringműveleteket függvényekkel valósították meg. Ezeket a függvényeket az

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

sor megadása után tudjuk használni.

- A `string.h` a felsoroltakon túl egyéb hasznos sztringműveleteket is tartalmaz. Linux alatt a `man string` parancs segítségével lehet ezekről információt kérni.



Sztring műveletek C-ben.

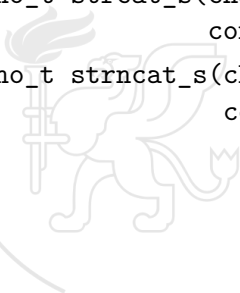
string.h fontosabb függvényei [1/2]

- `size_t strlen(const char *s)`
- `int strcmp(const char *s1, const char *s2)`
- `int strncmp(const char *s1, const char *s2, size_t n)`
- `char *strdup(const char *src)`
- `char *strndup(const char *src, size_t n)`
- `char *strcpy(char *dest, const char *src)`
- `char *strncpy(char *dest, const char *src, size_t n)`
- `char *strcat(char *dest, const char *src)`
- `char *strncat(char *dest, const char *src, size_t n)`

Sztring műveletek C-ben.

string.h fontosabb függvényei [2/2] (C11-től)

- `size_t strlen_s(const char *s, size_t sz)`
- `errno_t strcpy_s(char *dest, rsize_t sz, const char *src)`
- `errno_t strncpy_s(char *dest, rsize_t sz, const char *src, rsize_t n)`
- `errno_t strcat_s(char *dest, rsize_t sz, const char *src)`
- `errno_t strncat_s(char *dest, rsize_t sz, const char *src, rsize_t n)`



A string.h fontosabb függvényei.

strlen

- `size_t strlen(const char *s)`

Egy sztring hosszának meghatározása.

s A sztring.

returns A sztring hossza.

- Nem definiált a viselkedése arra az esetre, ha a kapott paraméter NULL;
- Le nem zárt sztring esetén buffer overflow-t (overread) okozhat.



A string.h fontosabb függvényei.

strcmp

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

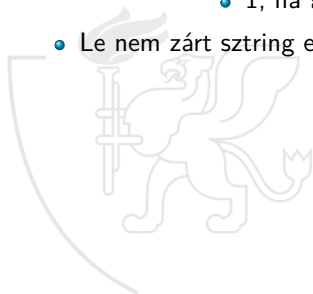
Két sztring összehasonlítása lexicografikus rendezéssel.

s1 Az első sztring.

s2 A második sztring.

returns

- -1, ha $s1 <_{lex} s2$;
 - 0, ha $s1 =_{lex} s2$;
 - 1, ha $s1 >_{lex} s2$.
- Le nem zárt sztring esetén buffer overflow-t (overread) okozhat.



A string.h fontosabb függvényei.

strncmp

- `int strncmp(const char *s1, const char *s2, size_t n)`
Két sztring összehasonlítása lexikografikus rendezéssel.

`s1` Az első sztring.

`s2` A második sztring.

`n` Az összehasonlításhoz használt karakterek maximális száma.

returns

- -1, ha $s1 <_{lex} s2$;
 - 0, ha $s1 =_{lex} s2$;
 - 1, ha $s1 >_{lex} s2$.
- Az összehasonlításhoz a sztringek első legfeljebb `n` karakterét használja, ha a két sztring addig egyezik, akkor egyenlőnek tekinti őket.
 - Helyesen használva véd a buffer overflow ellen.

A string.h fontosabb függvényei.

strdup

- `char *strdup(const char *src)`

Egy sztring duplikálása új sztring létrehozásával.

src A duplikálandó sztring.

returns Az új sztring pointere sikeres duplikálás esetén, különben a NULL pointer.

- Mivel a függvény memóiafoglalást végez, nem szabad megfeledezni a visszaadott pointer felszabadításáról, ha arra már nincs szükség;
- Le nem zárt sztring esetén buffer overflow-t (overread) okozhat.



A string.h fontosabb függvényei.

strndup

- `char *strndup(const char *src, size_t n)`

Egy sztring duplikálása új sztring létrehozásával.

src A duplikálandó sztring.

n A duplikált sztring maximális hossza.

returns Az új sztring pointere sikeres duplikálás esetén,
különben a NULL pointer.

- Az `strndup` esetén a duplikált sztring legfeljebb `n` hosszúságú lesz akkor is, ha az eredeti hosszabb volt;
- Mivel a függvény memóiafoglalást végez, nem szabad megfeledezni a visszaadott pointer felszabadításáról, ha arra már nincs szükség;
- Helyesen használva véd a buffer overflow ellen.

A string.h fontosabb függvényei.

strcpy

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

Egy sztring értékének másolása megadott helyre.

dest A cél sztring, ahová másolni kell.

src A forrás sztring, aminek az értékét másolni kell.

returns A dest pointer.

- Mivel nincs információja a sztringek méretéről, buffer overflow-t okozhat a használata.



A string.h fontosabb függvényei.

strncpy

- `char *strncpy(char *dest, const char *src, size_t n)`
Egy sztring értékének másolása megadott helyre.
 - dest** A cél sztring, ahová másolni kell.
 - src** A forrás sztring, aminek az értékét másolni kell.
 - n** Az átmásolandó karakterek maximális száma.
 - returns** A dest pointer.
- Ha a forrás hossza legalább `n`, akkor a cél sztring végére nem kerül lezáró karakter;
- Ha a forrás hossza rövidebb `n`-nél, akkor az `n`-ből fennmaradó helyet nulla kódú karakterekkel tölti fel;
- Mivel nincs információja a dest sztring méretéről, buffer overflow-t okozhat.

A string.h fontosabb függvényei.

strcat

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

Egy sztring hozzáfűzése egy másik sztringhez.

dest A bővítendő sztring.

src A hozzáfűzendő sztring.

returns A dest pointer.

- Garantált az eredménystring null karakterrel való lezárása;
- Mivel nincs információja a sztringek méretéről, buffer overflow-t okozhat a használata.



A string.h fontosabb függvényei.

strncat

- `char *strncat(char *dest, const char *src, size_t n)`

Egy sztring hozzáfűzése egy másik sztringhez.

dest A bővítendő sztring.

src A hozzáfűzendő sztring.

n A hozzáfűzendő sztring maximális hossza.

returns A dest pointer.

- Legfeljebb n karaktert fűz hozzá;
- Garantált az eredmény lezárása (akár az n+1 -ik karakterrel is);
- Mivel nincs információja a dest sztring méretéről, buffer overflow-t okozhat.

A string.h fontosabb függvényei.

strlen_s

- `size_t strnlen_s(const char *s, size_t sz)`

Egy sztring hosszának meghatározása (C11-től).

s A sztring.

sz A sztring mérete.

returns A sztring hossza, vagy 0, ha NULL pointert kapott, vagy sz, ha a sztring addig nincs lezárva.

- Kezeli a NULL sztringet is (ami nem azonos az üres sztringgel);
- Helyesen használva véd a buffer overflow ellen.

A string.h fontosabb függvényei.

strcpy_s

- `errno_t strcpy_s(char *dest, rsize_t sz, const char *src)`

Egy sztring értékének biztonságosabb másolása megadott helyre (C11-től).

dest A cél sztring, ahová másolni kell.

sz A cél sztring mérete.

src A forrás sztring, aminek az értékét másolni kell.

returns Sikeres művelet esetén 0, hiba esetén nem 0.

- Észreveszi, hogy a két sztring átlapoló, vagy ha valamelyik NULL;
- Észreveszi, ha a `dest` mérete nem nagyobb mint `src` hossza;
- Mivel nincs információja az `src` méretéről, buffer overflow-t (overread) okozhat.

A string.h fontosabb függvényei.

strncpy_s

- `errno_t strncpy_s(char *dest, rsize_t sz, const char *src, rsize_t n)`

Egy sztring értékének biztonságos másolása megadott helyre (`C11`-től).

dest A cél sztring, ahová másolni kell.

sz A cél sztring mérete.

src A forrás sztring, aminek az értékét másolni kell.

n Az átmásolandó karakterek maximális száma.

returns Sikeres művelet esetén 0, hiba esetén nem 0.

- A cél sztring mindenképpen lezárásra kerül, de ezen felül nem történik zero padding;
- Észreveszi, hogy a két sztring átlapoló, vagy ha valamelyik NULL;
- Észreveszi, ha a dest mérete nem nagyobb mint src hossza;
- Hiba esetén üres sztringet ad vissza;
- Helyesen használva véd a buffer overflow ellen.

A string.h fontosabb függvényei.

strcat_s

- `errno_t strcat_s(char *dest, rsize_t sz, const char *src)`

Egy sztring biztonságosabb hozzáfűzése egy másik sztringhez (C11-től).

dest A bővítendő sztring.

sz A bővítendő sztring mérete.

src A hozzáfűzendő sztring.

returns Sikeres művelet esetén 0, hiba esetén nem 0.

- Észreveszi, hogy a két sztring átlapoló, vagy ha valamelyik NULL;
- Észreveszi, ha a dest mérete nem nagyobb mint dest és src együttes hossza;
- Hiba esetén üres sztringet ad vissza;
- Garantált az eredménysztring null karakterrel való lezárása;
- Mivel nincs információja az src méretéről, buffer overflow-t (overread) okozhat.

A string.h fontosabb függvényei.

strncat_s

- `errno_t strncat_s(char *dest, rsize_t sz, const char *src, rsize_t n)`

Egy sztring biztonságos hozzáfűzése egy másik sztringhez (`C11` -től).

dest A bővítendő sztring.

sz A bővítendő sztring mérete.

src A hozzáfűzendő sztring.

n A hozzáfűzendő sztring maximális hossza.

returns Sikeres művelet esetén 0, hiba esetén nem 0.

- Észreveszi, hogy a két sztring átlapoló, vagy ha valamelyik NULL;
- Észreveszi, ha a `dest` mérete nem nagyobb mint `dest` és `src` együttes hossza;
- Hiba esetén üres sztringet ad vissza;
- Legfeljebb `n` karaktert fűz hozzá;
- Garantált az eredmény lezárása (akár az `n+1` -ik karakterrel is);
- Helyesen használva véd a buffer overflow ellen.

Sztring műveletek C-ben.

Hossz

- $\text{Hossz}(\rightarrow s : \text{Sztring}, \leftarrow i : N^+)$
 - A hossz-számítás meg van valósítva:

```
size_t strlen(const char *s);
```

```
i = strlen(s);
```



Sztring műveletek C-ben.

Értékadás

- $d = s$
 - Értékadásra a

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

függvény használható. A programozó felelőssége, hogy megfelelő méretű sztringeket használjon.

```
strcpy(d, s);
```



Sztring műveletek C-ben.

Összefűzés

- $\text{Összefűz}(\rightarrow s1 : \text{Sztring}, \rightarrow s2 : \text{Sztring}, \leftarrow s : \text{Sztring})$

- Az összefűzés a

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

segítségével történhet.

```
strcpy(s, s1);  
strcat(s, s2);
```



Sztring műveletek C-ben.

Egyenlőség

- Egyenlő($\rightarrow s1 : \text{Sztring}, \rightarrow s2 : \text{Sztring}$): Logikai
 - Két sztring egyenlőségének ellenőrzése az

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

lexikografikusan összehasonlító függvény segítségével történik.

```
(strcmp(s1, s2) == 0);
```



- 1 **Bemutakozás**
 - Kurzus információk
 - A SZTE és az informatikai képzés
- 2 **Linux**
 - Alapfogalmak
 - Linux parancsok
 - Linux shell
 - Felhasználók
 - Hálózat
- 3 **Gyors C áttekintés**
 - Bevezető
 - Pénzváltás (1. verzió)
 - Pénzváltás (2. verzió)
 - Röppálya számítás
 - Röppálya szimuláció
 - Az év napja
 - Csúszóátlag adott elemszámmra
 - Csúszóátlag parancssorból
 - Basename standard inputról
 - Basename parancssorból
 - Tér legtávolabbi pontjai
 - A nappalis gyakorlat értékelése

- 4 **Alapok**
 - Alapfogalmak
 - A programozás fázisai
 - Algoritmus vezérlése
 - A C nyelvű program
 - Szintaxis
 - A C nyelv elemi adattípusai
 - A C nyelv utasításai
- 5 **Vezérlési szerkezetek**
 - Bevezetés
 - Szekvenciális vezérlés
 - Függvények
 - Szelekciós vezérlések
 - Ismétléses vezérlések 1.
 - Eljárásvezérlés
 - Ismétléses vezérlések 2.
- 6 **Folyamatábra és struktúradiagram**
- 7 **Adatszerkezetek**
 - Az adatkezelés szintjei
 - Elemi adattípusok
 - Pointer adattípus
 - Tömb adattípus

- 8 **Sztringek**
- 9 **Pointerek és tömbök C-ben**
 - Rekord adattípus
 - Függvény pointer
 - Halmaz adattípus
 - Flexibilis tömbök
 - Láncolt listák
 - Típusokról C-ben
- 10 **IO**
 - Alapok
 - Adatállományok
- 9 **C fordítás**
 - A fordítás folyamata
 - A preprocesszor
 - A C fordító
 - Assembler
 - Linker és modulok
- 10 **Gyakorlati kérdések**
 - Memóriahasználat
 - Gyakori C hibák
 - where.c felboncolva

- 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.



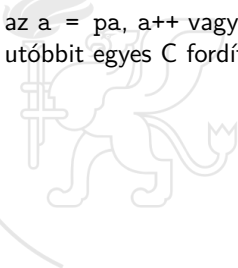
- Az `int a[10];` deklaráció definiál egy `10 * sizeof(int)` bájtos memóriaterületet, melynek egyes elemeire hivatkozhatunk az `a[0]`, `a[1]`, ..., `a[9]` változóhivatkozásokkal.
- Ha `pa` deklarációja `int *pa;` akkor a `pa = &a[0];` értékadás úgy állítja be `pa`-t, hogy az az a nulladik elemére mutasson, vagyis `pa` az `a[0]` elem címét tartalmazza.
- Ekkor az `x = *pa;` értékadás `a[0]` tartalmát másolja `x`-be.



- 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.
- Így ha pa az $a[0]$ -ra mutat, akkor
 - $*(pa+1)$ az $a[1]$ 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, mert a pointer aritmetika alapdefiníciója szerint a növekmény mértékegysége annak az objektumnak a tárbeli mérete, amire a mutató mutat.

- Az indexelés és a pointer aritmetika között láthatóan nagyon szoros kapcsolat van. Gyakorlatilag a tömbre való hivatkozást a fordító a tömb kezdetét megcímező mutatóvá alakítja át.
- Mivel a tömb neve tulajdonképpen egy mutatóérték, az illető tömb nulladik elemének címe, a $pa = \&a[0]$ értékadás írható egyszerűbben $pa = a$ -ként.
- Továbbá 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).
- Ha mindkét elemre alkalmazzuk az $\&$ operátort, akkor látható, hogy $\&a[i]$ és $a+i$ szintén azonosak: mindkettő az a -t követő i . elem címe.

- Másrészt, ha `pa` mutató, akkor azt kifejezésekben indexelhetjük tömbként `pa[i]` alakban, ami ugyanaz, mint a `*(pa+i)` kifejezés.
- 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 a `pa = a` vagy `pa++` értelmes műveletek.
 - A tömbnév azonban **nem** változóhivatkozás csak egy pointer *érték*, így az `a = pa`, `a++` vagy `p = &a` műveletek nem megengedettek (bár ez utóbbit egyes C fordítók, például a `gcc` is, megértik)!



Tömbök függvényparaméterként

- 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.
- 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.
 - Ezért például a sztringkezelő függvények definíciójában a paraméterként kapott sztring megadható
 - `char s[]`és
 - `char *s`formában is; azt, hogy adott esetben melyiket használjuk nagymértékben az dönti el, hogy a megvalósítás során miként írjuk le a kifejezéseket a függvényen belül.

Tömbök függvényparaméterként

- Lehetőség van arra, hogy a tömbnek csupán egy részét adjuk át valamelyik függvénynek oly módon, hogy a résztömb kezdetét megcímző mutatót adunk át.
 - Ha például a egy tömb neve, akkor az
 - `f(&a[2])`

és

 - `f(a+2)`

is az `a[2]` elem címét adja át a függvénynek. Ilyen hívás esetén az `f()` függvény az a tömb harmadik elemével kezdődő résztömböt kapja meg paraméterként, ezzel fog dolgozni.

- Az `f()` függvény deklarációja
 - `f(int arg[]) { ... }`

vagy 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.

Tömbök függvényparaméterként

Többsdimenziós tömbök

- Az egydimenziós tömb mérete elhagyható, de az elemének méretét már pontosan meg kell adni! Vagyis a sorok száma nem érdekes (első dimenzió mérete), de az oszlopok számát (második dimenzió mérete) meg kell adni a helyes címszámítás érdekében.

- Tekintsük például az

- `int a[5][35];`

alakban deklarált tömböt. A formális paraméter típusát

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

vagy

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

alakban is meg lehet adni.

- A zárójelezés szükséges, mert `[]` magasabb prioritású művelet, mint a `*` művelet, így

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

- `int *a[35];` egy `a` tömböt deklarál, ami 35 pointerből áll és a pointerok egészekre mutatnak.

- A következő példa talán rávilágít tömbök és mutatók között meglévő különbségre.
- Tekintsünk a következő deklarációkat:

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

- Szabályos változóhivatkozások a `honap[3][4]` és `Honap[3][4]` is, a megvalósítás viszont teljesen eltér a két esetben.
 - 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 összességében foglalhat több, vagy kevesebb memóriát is.

Pointerek és tömbök

```
char Honap[][20] = { "nincs_0._hónap",  
    "január", "február", ... , "december" };
```

n	i	n	c	s		0	.		h	ó	n	a	p	\0					
j	a	n	u	á	r	\0													
f	e	b	r	u	á	r	\0												
m	á	r	c	i	u	s	\0												
á	p	r	i	l	i	s	\0												
m	á	j	u	s	\0														
j	ú	n	i	u	s	\0													
j	ú	l	i	u	s	\0													
a	u	g	u	s	z	t	u	s	\0										
s	z	e	p	t	e	m	b	e	r	\0									
o	k	t	ó	b	e	r	\0												
n	o	v	e	m	b	e	r	\0											
d	e	c	e	m	b	e	r	\0											

```
char *honap[] = { "nincs_0._hónap",  
    "január", "február", ... , "december" };
```

[0] @0	n	i	n	c	s		0	.		h	ó	n							
[1] @15	a	p	\0	j	a	n	u	á	r	\0	f	e							
[2] @22	b	r	u	á	r	\0	m	á	r	c	i	u							
[3] @30	s	\0	á	p	r	i	l	i	s	\0	m	á							
[4] @38	j	u	s	\0	j	ú	n	i	u	s	\0	j							
[5] @46	ú	l	i	u	s	\0	a	u	g	u	s	z							
[6] @52	t	u	s	\0	s	z	e	p	t	e	m	b							
[7] @59	e	r	\0	o	k	t	ó	b	e	r	\0	n							
[8] @66	o	v	e	m	b	e	r	\0	d	e	c	e							
[9] @76	m	b	e	r	\0														
[10] @87																			
[11] @95																			
[12] @104																			

- A pointeraritmetika alapja, hogy a pointert mindig az általa mutatott típus méretének függvényében változtatjuk.
- A pointerok és egész számok közötti összeadás/kivonás alpműveletet már láttuk és értelmeztük.
- Az alpműveletek alapján több műveletet is értelmezhetünk:
 - A $p += i$ és $p -= i$ műveleteket az egész műveletekhez hasonló módon értelmezhetjük, azaz ha a p egy tömbelemre mutatott, akkor a művelet hatására az azt követő illetve megelőző i . elemre fog mutatni.
 - A $++$ vagy $--$ művelet p -re alkalmazása után p a következő vagy előző tömbelemre mutat majd. Magának a műveletnek az eredménye a prefix vagy postfix alak alkalmazásától függ.
- Mutatók kivonása szintén megengedett: ha p és q azonos típusú mutatók, akkor $p - q$ a két mutató közötti tömbelemek (előjeles) darabszámát adja. (Arra persze a programozónak kell figyelnie, hogy ez *legális* érték-e, vagyis p és q valóban ugyanannak a tömbnek két elemére mutat-e?)

- A pointeraritmetika `void*` típus esetén is működik.
- Mivel a `void` típus önmagában 0 bájton (nem) tárolódik, a pointeraritmetikai számítások alapegysége definíció szerint 1 bájtt.



- 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.



- 1 **Bemutakozás**
 - Kurzus információk
 - A SZTE és az informatikai képzés
- 2 **Linux**
 - Alapfogalmak
 - Linux parancsok
 - Linux shell
 - Felhasználók
 - Hálózat
- 3 **Gyors C áttekintés**
 - Bevezető
 - Pénzváltás (1. verzió)
 - Pénzváltás (2. verzió)
 - Röppálya számítás
 - Röppálya szimuláció
 - Az év napja
 - Csúszóátlag adott elemszámmra
 - Csúszóátlag parancssorból
 - Basename standard inputról
 - Basename parancssorból
 - Tér legtávolabbi pontjai
 - A nappalis gyakorlat értékelése

- 4 **Alapok**
 - Alapfogalmak
 - A programozás fázisai
 - Algoritmus vezérlése
 - A C nyelvű program
 - Szintaxis
 - A C nyelv elemi adattípusai
 - A C nyelv utasításai
- 5 **Vezérlési szerkezetek**
 - Bevezetés
 - Szekvenciális vezérlés
 - Függvények
 - Szelekciós vezérlések
 - Ismétléses vezérlések 1.
 - Eljárásvezérlés
 - Ismétléses vezérlések 2.
- 6 **Folyamatábra és struktúradiagram**
- 7 **Adatszerkezetek**
 - Az adatkezelés szintjei
 - Elemi adattípusok
 - Pointer adattípus
 - Tömb adattípus

- 8 **Sztringek**
 - Pointerek és tömbök C-ben
- 9 **Rekord adattípus**
 - Függvény pointer
 - Halmaz adattípus
 - Flexibilis tömbök
 - Láncolt listák
 - Típusokról C-ben
- 10 **IO**
 - Alapok
 - Adatállományok
- 9 **C fordítás**
 - A fordítás folyamata
 - A preprocesszor
 - A C fordító
 - Assembler
 - Linker és modulok
- 10 **Gyakorlati kérdések**
 - Memóriahasználat
 - Gyakori C hibák
 - where.c felboncolva

- 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 absztrakt adattípus

- Ha az egyes típusú adatokat egyszerre kell tudnunk tárolni, szorzat-rekordról beszélünk.
- Legyenek T_1, \dots, T_k tetszőleges típusok.
- Ezekből képezzük a
$$T = T_1 \times \dots \times T_k = \{(a_1, \dots, a_k) \mid a_1 \in T_1, \dots, a_k \in T_k\}$$
értékhalmoz, tehát a T_1, \dots, T_k típusok értékhalmozainak direktszorzatát.
- A T halmazon is értelmezhetünk kiolvasó és módosító műveletet, mint a tömb típus esetén, de a k eltérő típus miatt most k számú kiolvasó és módosító műveletre lesz szükségünk.
- Az új adattípusra a $T = \text{Rekord}(T_1, \dots, T_k)$ jelölést használjuk és szorzat-rekordnak vagy struktúrának nevezzük.

Szorzat-rekord absztrakt adattípus műveletei

- $\text{Kiolvas}_i(\rightarrow A:T; \leftarrow x:T_i)$
 - Adott $i \in \{1 \dots k\}$ -ra az A rekord i . komponensének kiolvasása adott x , T_i típusú változóba.
- $\text{Módosít}_i(\leftrightarrow A:T; \rightarrow x:T_i)$
 - Adott $i \in \{1 \dots k\}$ -re az A rekord i . komponensének módosítása adott x , T_i típusú értékre.
- $\text{Értékadás}(\leftarrow A:T; \rightarrow X:T)$
 - Értékadó művelet. Az A változó felveszi az X , T típusú kifejezés értékét.



A struct virtuális adattípus

- A $T = \text{Rekord}(T_1, \dots, T_k)$ típust C-ben a struct kulcsszóval definiáljuk:

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

- A fenti típusképzésben az M_1, \dots, M_k azonosítókat mezőazonosítóknak (tagnak, member-nek) hívjuk és lokálisak a típusképzésre nézve.

A struct virtuális adattípus

- Az absztrakt típus műveletei mezőhivatkozások segítségével valósíthatók meg.
- A mezőhivatkozásra a `.` operátort használjuk. Ez egy olyan rekordokon értelmezett művelet C-ben, ami nagyon magas precedenciával rendelkezik és balasszociatív.
- Egy rekordra a mezőkiválasztás operátort (megfelelő mezőnévvel) alkalmazva a rekord mezőjét változóként kapjuk vissza.



C műveletek prioritása

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

Szorzat-rekord

Megvalósítás C nyelven

- $\text{Kiolvas}_i(\neg A:T; \leftarrow x:T_i)$

```
x = A.Mi
```

- $\text{Módosít}_i(\leftrightarrow A:T; \rightarrow x:T_i)$

```
A.Mi = x
```

- $\text{Értékadás}(\leftarrow A:T; \rightarrow X:T)$

```
A = X
```

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;

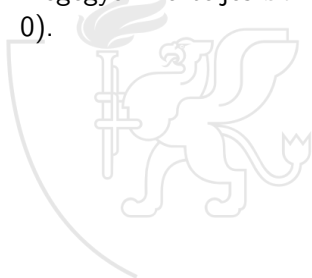
typedef struct SzemelyTip {
    struct {
        Szoveg20 családi, uto;
    } nev;
    int szemelyiSzam;
    Szoveg20 szulHely;
    DatumTip szulIdo;
    CimTip lakcim;
} SzemelyTip;
```

- SzemelyTip x; deklaráció esetén például az x.nev.csaladi és x.szulIdo.ev érvényes mezőhivatkozások.

Szorzat-rekord / struct

Fizikai adattípus

- A struct típusú változó számára foglalt memória mérete, amely a `sizeof` függvénnyel lekérdezhető:
 - $\text{sizeof}(E) = \text{sizeof}(T_1) + \dots + \text{sizeof}(T_k) + \text{igazítás}$
- Valamennyi változati mező a deklaráció sorrendjében egymást (a mezők méretét és az esetleges igazítást is figyelembe véve) követő, növekvő memóriacímen kezdődik. Az első mező memóriacíme megegyezik a teljes struct típusú érték címével (az eltolása, offset-je 0).



Eltelt idő kiszámítása [1/2]

eltelt-struct.c [1-25]

```
1 /* Egy nap két időpontja között mennyi idő telt el.
2  * 2013. November 7. Gergely Tamás, gertom@inf.u-szeged.hu
3  */
4
5 #include <stdio.h>
6
7 typedef struct ido_t {
8     int ora;
9     int perc;
10 } ido_t;
11
12 int ido_to_int(ido_t t) {
13     return 60 * t.ora + t.perc;
14 }
15
16 ido_t int_to_ido(int t) {
17     ido_t ret;
18     ret.ora = t / 60;
19     ret.perc = t % 60;
20     return ret;
21 }
22
23 ido_t eltelt_ido(ido_t t1, ido_t t2) {
24     return int_to_ido(ido_to_int(t2) - ido_to_int(t1));
25 }
```

Eltelt idő kiszámítása [2/2]

eltelt-struct.c [27–39]

```
27 int main() {
28     ido_t t1, t2, dt;
29     /* beolvasás */
30     printf("Kérem az első időpontot óra perc formában\n");
31     scanf("%d %d", &t1.ora, &t1.perc);
32     printf("Kérem a második időpontot óra perc formában\n");
33     scanf("%d %d", &t2.ora, &t2.perc);
34     /* számítás */
35     dt = eltelt_ido(t1, t2);
36     /* kiírás */
37     printf("Az eltelt idő: %d óra %d perc.\n", dt.ora, dt.perc);
38     return 0;
39 }
```



Egyesített-rekord absztrakt adattípus

- Ha az egyes típusú adatokat nem kell egyszerre tárolni, egyesített-rekordról beszélünk.
- Legyenek $T_0 = \{c_1, \dots, c_k\}$ és T_1, \dots, T_k tetszőleges típusok.
- Ezekből képezzük a
$$T = T_1 + \dots + T_k = \cup_{c_i \in T_0} (\{c_i\} \times T_i) = \{(i, a) | i \in T_0, a \in T_i\}$$
értékhalmozat, tehát a T_1, \dots, T_k típusok értékhalmozainak T_0 szerinti diszjunkt egyesítését. T elemei tehát olyan rendezett párok, amelyeknek első komponense meghatározza, hogy a második komponens melyik típusból való érték.
- A T halmazon is a szorzat rekordhoz hasonló módon értelmezhetünk kiolvasó és módosító műveletet.
- Az új adattípust a T_0 változati típusból és T_1, \dots, T_k egyesítési-tag típusokból képzett egyesített-rekord típusnak nevezzük.

Egyesített-rekord absztrakt adattípus műveletei

- $\text{Változat}(\rightarrow A:T; \leftarrow V:T_0)$
 - Változat kiolvasása. A művelet végrehajtása után $V = c_i$, ha $A = (c_i, a)$.
- $\text{Kiolvas}_i(\rightarrow A:T; \leftarrow x:T_i)$
 - Adott $i \in \{1 \dots k\}$ -ra az A rekord i . komponensének kiolvasása adott x , T_i típusú változóba. A művelet végrehajtása után $x = a$, ha $A = (c_i, a)$. A művelet hatástalan, ha A első komponense nem c_i .
- $\text{Módosít}_i(\leftarrow A:T; \rightarrow x:T_i)$
 - Adott $i \in \{1 \dots k\}$ -re az A rekord i . komponensének módosítása adott x , T_i típusú értékre. A művelet végrehajtása után $A = (c_i, x)$.
- $\text{Értékadás}(\leftarrow A:T; \rightarrow X:T)$
 - Értékadó művelet. Az A változó felveszi az X , T típusú kifejezés értékét.

Az union virtuális adattípus

- Az egyesített-rekord típust C-ben az `union` kulcsszó segítségével valósíthatjuk meg.
- Az `union` kulcsszó nagyon hasonlít a `struct`-hoz:

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

- A fenti típusképzésben az M_1, \dots, M_k azonosítókat mezőazonosítóknak (tagnak, member-nek) hívjuk és lokálisak a típusképzésre nézve.

Az union virtuális adattípus

- Az absztrakt típus műveletei itt is mezőhivatkozások (. operátor) segítségével valósíthatóak meg, ami ugyanúgy működik mint a struct típusnál.
- Megjegyzendő, hogy a C megvalósításában (megfelelő környezetben) mindig hivatkozhatunk bármelyik mezőre, függetlenül attól, hogy az union aktuálisan melyik mező értékét tárolja.
- A C union típusképzésében nem adhatunk meg változati mezőazonosítót (T_0), így nincs lehetőségünk az aktuális változatról információ tárolására.

- Mint látható, az `union` konstrukcióban nincs lehetőség a jelzőmező (T_0) megadására, ezért ha szükségünk van a jelzőmezőre is, akkor a C megvalósításhoz kombinálnunk kell a `struct` és `union` típusképzéseket.

```
typedef struct T {  
    T0 Milyen;  
    union {  
        T1 M1;  
        ...  
        Tk Mk;  
    };  
} T;
```

- Jelzőmezőnek felsorolás (vagy valamilyen egész) típust érdemes használni.

Egyesített-rekord

Megvalósítás C nyelven

- Változat($\rightarrow A:T$; $\leftarrow V:T_0$)

```
V = A.Milyen
```

- Kiolvas_i($\rightarrow A:T$; $\leftarrow x:T_i$)

```
if (A.Milyen == ci) { x = A.Mi; }
```

- Módosít_i($\leftrightarrow A:T$; $\rightarrow x:T_i$)

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

- Értékadás($\leftarrow A:T$; $\rightarrow X:T$)

```
A = X
```

Egyesített-rekord

Példák

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

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

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;
    Sikidom forma;
    union {
        double sugar;
        struct { double alfa, oldal1, oldal2; };
        struct { double hossz, szel; };
    };
} Alakzat;
```

Egyesített-rekord / union

Fizikai adattípus

- A union típusú változó számára foglalt memória mérete, amely a `sizeof` függvénnyel lekérdezhető:
 - $\text{sizeof}(T) = \max\{\text{sizeof}(T_1), \dots, \text{sizeof}(T_k)\}$
- Valamennyi változati mező ugyanazon a memóriacímen kezdődik, ami megegyezik a teljes union típusú érték címével (azaz minden mező eltolása, offset-je 0).



A struct és union típusok

Egymásba ágyazás

- Mint az a példából 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 */
```



A struct és union típusok

Struktúracímke

- A struct vagy az union kulcsszót struktúracímke (struktúranév) követheti.
 - Ez egy név, amely megnevezi az adott típusú struktúrát, vagy uniont és a továbbiakban rövidítésként használható a {}-ben lévő részletes deklaráció helyett.
 - Struktúranevet a típus megadásakor attól függetlenül megadhatunk, hogy a típust éppen hogyan használjuk fel.
- A struktúranév és a typedef-fel megadott típusnév meg is egyezhet (mint a fenti példákban), így a legrugalmasabb a felhasználásuk. Pl.:

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

A struct és union típusok

Műveletek

- A struct és az union típusú változók esetén megengedett az értékadás művelet, így függvényargumentumként is használhatjuk ezeket (érték szerinti paraméterátadás) illetve a függvényművelet eredményének típusa is lehet struct vagy union.
- Megengedett az & művelet használata is, így a struct és az union típusú változók cím szerinti paraméterátadással is kezelhetők.



A struct és union típusok

Inicializálás

- A struct és az union típusú változók is kaphatnak kezdőértéket, az adattagok értékeit { és } zárójelek között megadva.
- Értékek szimpla felsorolásával
 - egy struct-ban az adattagok sorrendben inicializálhatók, de nem kötelező mindegyiket inicializálni;
 - az union típusnak csak a deklaráció szerinti első tagja inicializálható.

```
DatumTip d = {1970, 1, 1};  
Alakzat egysegKor = { 0, 0, kor, { 1.0 } };
```

- A C99 szabvány lehetővé teszi tetszőleges mezők inicializálását a struct és union típusok esetén is. Ekkor a {} zárójelek között az egyes értékek elé .mezo =-t írva jelezhetjük, hogy az adott érték a mezo mező kezdőértéke lesz.

```
Idom i = { .UU = { .U1 = { .C = 3, .B = 4, .A = 5 } }, .Fajta = haromszog };
```

- A C nyelv lehetővé teszi egy byte-on belüli bitek elérését magas szinten, bitmezős struktúrák segítségével.
- Felhasználhatjuk pl. hardver-programozáshoz szükséges bitsorozatok magas szintű kezelésére (driverok írásához), vagy jelzőbitek (flag-ek) tömör elhelyezésére.
- A fizikai megvalósítás gépfüggő.



- A bitmezőket a struktúrákon belül használhatjuk (akár a *normális* mezőkkel kombinálva is). A különbség az, hogy meg kell adni a tároláshoz szükséges bitek számát.

```
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) {  
    /* ... */  
}
```

- Miért jó ez? Bitmezők nélkül az alábbi struktúra legalább 6 bájt lenne, így viszont csak 2.

```
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;
```



Síkidomok területe és kerülete

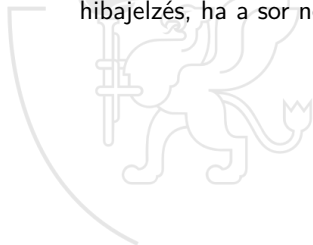
Problémafelvetés és specifikáció

- Problémafelvetés

- Határozzuk meg kör, téglalap és háromszög alakú síkidomok területét és kerületét.

- Specifikáció

- A probléma inputja több sor „ $\text{kor}(r)$ ”, „ $\text{teglalap}(a,b)$ ” vagy „ $\text{haromszog}(a,b,c)$ ” alakú szöveggel, ahol r , a , b és c valós számok. Az inputot a fájl vége (ctrl+D) zárja.
- Az output a megadott síkidom területe és kerülete két sorban, vagy egy hibajelzés, ha a sor nem megfelelő formátumú.



- Algoritmustervezés:
 - Az inputot soronként olvassuk be, és ha megfelel valamelyik síkidom leírásának, akkor feltöltünk egy síkidom adattípust (`union`-ok és `struct`-ok kombinációja) a megfelelő adatokkal.
 - A fő algoritmusban csak az input adatokat olvassuk be, majd meghívjuk a terület és kerületszámító függvényeket, és kiírjuk a kiszámolt értékeket.
 - A terület- és kerületszámítást egy-egy függvény végezze, ami egy síkidomot kap.



Síkidomok 1. verzió [1/4]

sikidomok.c [1–24]

```
1 /* Síkidomok területének és kerületének kiszámítása.
2  * 2018. Szeptember 19. Gergely Tamás, gertom@inf.u-szeged.hu
3  */
4
5 #include <math.h>
6 #include <stdio.h>
7
8 #define PUFFERMERET 128
9
10 enum sikidom_tipus_t {
11     kor, haromszog, teglalap
12 };
13
14 struct kor_tulajdonsagok_t {
15     double r;
16 };
17
18 struct haromszog_tulajdonsagok_t {
19     double a, b, c;
20 };
21
22 struct teglalap_tulajdonsagok_t {
23     double a, b;
24 };
```

Síkidomok 1. verzió [2/4]

sikidomok.c [26–54]

```
26 struct sikidom {
27     enum sikidom_tipus_t tipus;
28     union {
29         struct kor_tulajdonsagok_t kor;
30         struct haromszog_tulajdonsagok_t haromszog;
31         struct teglalap_tulajdonsagok_t teglalap;
32     };
33 };
34
35 double kor_kerulet(struct kor_tulajdonsagok_t k) {
36     return 2.0 * M_PI * k.r;
37 }
38
39 double kor_terulet(struct kor_tulajdonsagok_t k) {
40     return M_PI * k.r * k.r;
41 }
42
43 double haromszog_kerulet(struct haromszog_tulajdonsagok_t h) {
44     return h.a + h.b + h.c;
45 }
46
47 double haromszog_terulet(struct haromszog_tulajdonsagok_t h) {
48     double s = (h.a + h.b + h.c) / 2.0;
49     return sqrt(s * (s-h.a) * (s-h.b) * (s-h.c));
50 }
51
52 double teglalap_kerulet(struct teglalap_tulajdonsagok_t t) {
53     return 2.0 * (t.a + t.b);
54 }
```

Síkidomok 1. verzió [3/4]

sikidomok.c [56–76]

```
56 double teglalap_terulet(struct teglalap_tulajdonsagok_t t) {
57     return t.a * t.b;
58 }
59
60 double kerulet(struct sikidom s) {
61     switch(s.tipus) {
62     case kor:         return kor_kerulet(s.kor);
63     case haromszog:   return haromszog_kerulet(s.haromszog);
64     case teglalap:    return teglalap_kerulet(s.teglalap);
65     default:          return NAN;
66     }
67 }
68
69 double terület(struct sikidom s) {
70     switch(s.tipus) {
71     case kor:         return kor_terulet(s.kor);
72     case haromszog:   return haromszog_terulet(s.haromszog);
73     case teglalap:    return teglalap_terulet(s.teglalap);
74     default:          return NAN;
75     }
76 }
```

Síkidomok 1. verzió [4/4]

sikidomok.c [78–103]

```
78 char puffer[PUFFERMERET];
79
80 int main() {
81     double a,b,c;
82     struct sikidom s;
83     while(fgets(puffer, PUFFERMERET, stdin)) {
84         if(sscanf(puffer, "kor(%lf)", &a) == 1) {
85             s.tipus = kor;
86             s.kor.r = a;
87         } else if(sscanf(puffer, "teglalap(%lf,%lf)", &a, &b) == 2) {
88             s.tipus = teglalap;
89             s.teglalap.a = a;
90             s.teglalap.b = b;
91         } else if(sscanf(puffer, "haromszog(%lf,%lf,%lf)", &a, &b, &c) == 3) {
92             s.tipus = haromszog;
93             s.haromszog.a = a;
94             s.haromszog.b = b;
95             s.haromszog.c = c;
96         } else {
97             printf("Ismeretlen_\formatumu_\sor!\n");
98             continue;
99         }
100         printf("T=_%lf\nK=_%lf\n", terület(s), kerulet(s));
101     }
102     return 0;
103 }
```

A struct, union és enum típusok

Deklaráció és definíció

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



A struct, union és enum típusok

Deklaráció és definíció

- Változódeklarációk:

```
struct {  
    int a, b;  
} vstruct;  
  
union {  
    long long l;  
    double d;  
} vunion;  
  
enum {  
    nulla, egy, ketto, harm  
} venum;
```



A struct, union és enum típusok

Deklaráció és definíció

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

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

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

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

- 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 {  
    int a, b;  
};
```

A struct, union és enum típusok

Deklaráció és definíció

- Változódeklaráció helyett készíthetünk típusdefiníciót is:

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

- Akár így:

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

- Vagy így:

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


A struct, union és enum típusok

Deklaráció és definíció

- A típusnév és a struktúranév viszont nem ugyanaz az dolog! Az előző deklarációk esetén az alábbiak hibásak:

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

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

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



- Az alapvetőbb típuskonstrukciók megismerése után visszatérhetünk a dinamikus változóknál megemlített három fogalomhoz:
 - változóhivatkozás;
 - hivatkozott változó;
 - változó értéke.
- A változóhivatkozás szintaktikus egység, tehát meghatározott formai szabályok szerint képzett jelsorozat egy adott programnyelven.
- A C nyelvben a változóhivatkozás neve *l-value*.
 - Értékadás bal oldalán vagy például a ++ operátor operandusaként csak ilyen l-value szerepelhet.
- 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.

- Az, hogy mely kifejezések tekinthetők (szintaktikailag) érvényes változóhivatkozásnak, az alábbi táblázat alapján dönthető el.

T	x	x	x.m	*x	x[i]	&x	⊙
char, int, enum, float, double	T x;	T	-	-	-	(T*)	(?)
	(T)e	(T)	-	-	-	-	(?)
struct {E m;}, union {E m;}	T x;	T	E	-	-	(T*)	(?)
	(T)e	(T)	(E)	-	-	-	(?)
∀	T* x;	T*	-	T	T	(T**)	(?)
	(T*)e	(T*)	-	T	T	-	(?)
	T x[];	(T*)	-	T	T	-	(?)
	(T[])e	(T*)	-	T	T	-	(?)

T: T típusú változóhivatkozás; (T): T típusú érték; (?): érték, a típus a ⊙ művelettől függ.

- 1 **Bemutakozás**
- Kurzus információk
 - A SZTE és az informatikai képzés

- 2 **Linux**
- Alapfogalmak
 - Linux parancsok
 - Linux shell
 - Felhasználók
 - Hálózat

- 3 **Gyors C áttekintés**
- Bevezető
 - Pénzváltás (1. verzió)
 - Pénzváltás (2. verzió)
 - Röppálya számítás
 - Röppálya szimuláció
 - Az év napja
 - Csúszóátlag adott elemszámmra
 - Csúszóátlag parancssorból
 - Basename standard inputról
 - Basename parancssorból
 - Tér legtávolabbi pontjai
 - A nappalis gyakorlat értékelése

- 4 **Alapok**
- Alapfogalmak
 - A programozás fázisai
 - Algoritmus vezérlése
 - A C nyelvű program
 - Szintaxis
 - A C nyelv elemi adattípusai
 - A C nyelv utasításai

- 5 **Vezérlési szerkezetek**
- Bevezetés
 - Szekvenciális vezérlés
 - Függvények
 - Szelekciós vezérlések
 - Ismétléses vezérlések 1.
 - Eljárásvezérlés
 - Ismétléses vezérlések 2.

- 6 **Folyamatábra és struktúradiagram**
- 7 **Adatszerkezetek**
- Az adatkezelés szintjei
 - Elemi adattípusok
 - Pointer adattípus
 - Tömb adattípus

- Sztringek
- Pointerek és tömbök C-ben
- Rekord adattípus
- **Függvény pointer**
- Halmaz adattípus
- Flexibilis tömbök
- Láncolt listák
- Típusokról C-ben

- 8 **IO**
- Alapok
 - Adatállományok

- 9 **C fordítás**
- A fordítás folyamata
 - A preprocesszor
 - A C fordító
 - Assembler
 - Linker és modulok

- 10 **Gyakorlati kérdések**
- Memóriahasználat
 - Gyakori C hibák
 - where.c felboncolva

Függvényre mutató pointer

- Eddig kétféleképpen közelítettük meg a pointer típusú változót:
 - Első közelítésben a p pointer típusú változó értéke egy meghatározott típusú dinamikus változó.
 - A másik szerint a p pointer típusú változó értéke egy másik változóhoz tartozó memóriacím.
- A harmadik megközelítésben a pointer egy részalgoritmusra is mutathat, amelyet aktuális argumentumokkal végrehajtva a megadott típusú eredményhez jutunk.
 - 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 függvényfejlécben a függvény azonosítóját pointerre cseréljük, vagyis a típus azonosítója elé *-ot írunk.

Függvényre mutató pointer

- Mivel a `*` alacsonyabb prioritású, mint a `()` (függvényhívás művelet), ezért a dereferenciát zárójelpárba kell tenni.
- Legyen adott egy függvény, pl.:

```
double sin2x(double x)
{
    return sin(2.0 * x);
}
```

- Egy ilyen függvényre mutató pointer típus definíciója:

```
typedef double (*fuggveny_t)(double);
```

- És a használata:

```
fuggveny_t f = sin2x;
double x, y;
y = (*f)(x); /* Így jobban látszik, hogy f nem egy konkrét függvénynév, */
x = f(y);    /* de így is használható. */
```

Határozott integrál kiszámítása

Problémafelvetés és specifikáció

- Problémafelvetés

- Az $\frac{e^x}{x}$ és a $\sin(2x)$ függvények határozott integrálját közelítsük egy beolvasott intervallumon a felhasználó által megadott számú részre osztva az intervallumot!

- Specifikáció

- A probléma inputja a , b valós számok, az intervallum végpontjai, n , a közelítéshez használandó részek száma.
- Az output egy valós szám, a határozott integrál értéke.



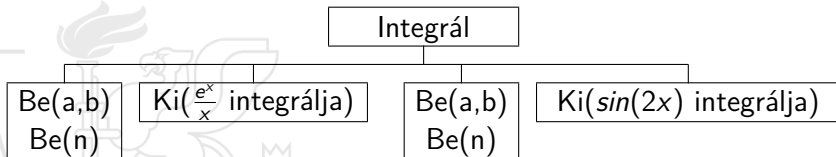
Határozott integrál kiszámítása

Algoritmustervezés

- Algoritmustervezés:

- Készítsünk egy függvényt, ami egy tetszőleges függvénynek képes kiszámolni az integrálját.
- A fő algoritmusban csak az input adatokat olvassuk be, majd meghívjuk az integráló függvényt az aktuális argumentumokkal, végül kiíratjuk az eredményt.

- Szerkezeti ábra:



Határozott integrált számító függvény

Problémafelvetés és specifikáció

- Problémafelvetés

- A paraméterként kapott függvény határozott integrálját közelítsük egy adott intervallumon adott számú részre osztva az intervallumot!

- Specifikáció

- A probléma inputja az integrálandó f függvény, az a , b valós számok mint az intervallum végpontjai, és n mint a közelítéshez használandó részek száma.
- Az output egy valós szám, a függvény $[a, b]$ intervallumon vett határozott integráljának közelítő értéke.

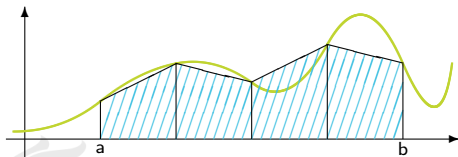


Határozott integrált számító függvény

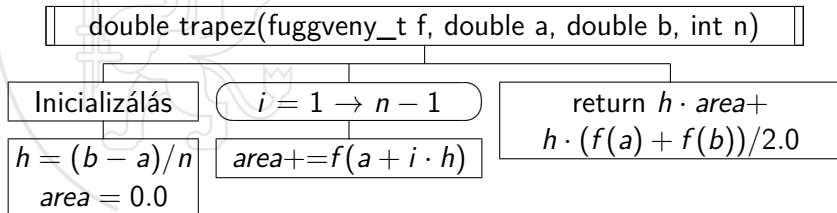
Algoritmustervezés

- Algoritmustervezés

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



- Szerkezeti ábra:



Határozott integrál kiszámítása [1/3]

integral.c [1-24]

```
1 /* Közelítő integrálás a trapéz szabály segítségével.
2  * Az integrálandó függvényt paraméterként kapjuk.
3  * 1998. Április 14. Dévényi Károly, devenyi@inf.u-szeged.hu
4  * 2006. Augusztus 14. Gergely Tamás, gertom@inf.u-szeged.hu
5  */
6
7 #include <stdio.h>
8 #include <math.h>
9
10 /* Az integrálandó függvény típusa */
11 typedef double (*fuggveny_t)(double x);
12
13 static double trapez(fuggveny_t f,          /* f(x)-t integráljuk az */
14                     double a, double b,    /* a,b intervallumon */
15                     int n) {               /* n részre osztva az interv.-t */
16     /* Közelítő integrálás a trapéz szabály segítségével. */
17     const double h = (b - a) / n;
18     double area = 0.0;
19     int i;                                /* a ciklusváltozó */
20
21     for (i = 1; i < n; ++i) {             /* fgv. értékek összegzése */
22         area += (*f)(a + i * h);          /* f(a + i * h) */
23     }
24     return (area * h + ((*f)(a) + (*f)(b)) / 2.0 * h); /* (f(a) + f(b)) */
25 }
```

Határozott integrál kiszámítása [2/3]

integral.c [26–34]

```
26 static double expx(double x) {  
27     /* az első integrálandó függvény */  
28     return (exp(x) / x);  
29 }  
30  
31 static double sin2x(double x) {  
32     /* az második integrálandó függvény */  
33     return sin(2.0 * x);  
34 }
```



Határozott integrál kiszámítása [3/3]

integral.c [36–58]

```
36 int main() {
37     double a, b;
38     int n;
39
40     printf("Az exp(x)/x függvény közelítő integrálja.\n");
41     printf("Kérem az integrálási intervallumot");
42     printf("és az osztáspontok számát(a,b,n)!\n");
43     printf("a:"); scanf("%lg%[\n]", &a); getchar();
44     printf("b:"); scanf("%lg%[\n]", &b); getchar();
45     printf("n:"); scanf("%d%[\n]", &n); getchar();
46     printf("Az integrál közelítő értéke:");
47     printf("%10.5f\n", trapez(expx, a, b, n));
48
49     printf("Az sin(2x) függvény közelítő integrálja.\n");
50     printf("Kérem az integrálási intervallumot");
51     printf("és az osztáspontok számát(a,b,n)!\n");
52     printf("a:"); scanf("%lg%[\n]", &a); getchar();
53     printf("b:"); scanf("%lg%[\n]", &b); getchar();
54     printf("n:"); scanf("%d%[\n]", &n); getchar();
55     printf("Az integrál közelítő értéke:");
56     printf("%10.5f\n", trapez(sin2x, a, b, n));
57     return 0;
58 }
```

- A C nyelvet támogató környezetekben lehetőség van arra, hogy a végrehajtás megkezdésekor a programnak parancssori argumentumokat vagy paramétereket adjunk át.
- Amikor az operációs rendszer elindítja a programot, azaz meghívja a `main` függvényt, a hívásban két argumentum szerepelhet:
 - Az első (általában `argc`) azoknak a parancssori argumentumoknak a darabszáma, amelyekkel a programot meghívtuk.
 - A második argumentum (általában `argv`) egy mutató egy sztring-tömbre, amely a parancssori argumentumokat tartalmazza. Egy karakterlánc egy argumentumnak felel meg.
 - Megállapodás szerint `argv[0]` az a név, amellyel a programot hívták, így az `argc` értéke legalább 1.
 - Számíthatunk arra is, hogy `argv[argc]==NULL`.
- Mivel az `argv` egy mutatótömböt megcímző mutató, a `main` függvényt többféleképpen deklarálhatjuk.

Parancssori argumentumok kezelése [1/1]

arg.c [1-21]

```
1 /* Kiírjuk a parancssorban lévő argumentumokat.
2  * 1998. Április 16. Dévényi Károly, devenyi@inf.u-szeged.hu
3  */
4
5 #include <stdio.h>
6
7 int main(int argc, char **argv) {
8
9     /* vagy így is lehet az argv-t deklarálni
10
11 int main(int argc, char *argv[]) {
12
13 */
14     int i;
15
16     printf("argc=%d\n\n", argc);
17     for (i = 0; i < argc; ++i) {
18         printf("argv[%d]: %s\n", i, argv[i]);
19     }
20     return 0;
21 }
```

- Mentsük el a fenti programot `arg.c` néven, fordítsuk le és futtassuk felparaméterezve!

```
$ gcc -o arg arg.c
$ ./arg alma korte szilva barack palinka
argc = 6
argv[0]: ./arg
argv[1]: alma
argv[2]: korte
argv[3]: szilva
argv[4]: barack
argv[5]: palinka
```



Eltelt idő kiszámítása [1/2]

eltelt-arg.c [1–25]

```
1 /* Egy nap két időpontja között mennyi idő telt el.
2  * 2013. November 7. Gergely Tamás, gertom@inf.u-szeged.hu
3  */
4
5 #include <stdio.h>
6 #include <stdlib.h>
7
8 typedef struct ido_t {
9     int ora;
10    int perc;
11 } ido_t;
12
13 ido_t str_to_ido(const char * str) {
14     ido_t ret = {0, 0};
15     ret.ora = atoi(str);
16     for (; *str && *str != ':'; ++str);
17     if (*str) {
18         ret.perc = atoi(str + 1);
19     }
20     return ret;
21 }
22
23 int ido_to_int(ido_t t) {
24     return 60 * t.ora + t.perc;
25 }
```

Eltelt idő kiszámítása [2/2]

eltelt-arg.c [27–48]

```
27 ido_t int_to_ido(int t) {
28     ido_t ret;
29     ret.ora = t / 60;
30     ret.perc = t % 60;
31     return ret;
32 }
33
34 ido_t eltelt_ido(ido_t t1, ido_t t2) {
35     return int_to_ido(ido_to_int(t2) - ido_to_int(t1));
36 }
37
38 int main(int argc, char *argv[]) {
39     ido_t t1, t2, dt;
40     if (argc < 3) {
41         return 1;
42     }
43     t1 = str_to_ido(argv[1]);
44     t2 = str_to_ido(argv[2]);
45     dt = eltelt_ido(t1, t2);
46     printf("Az eltelt idő %d:%02d\n", dt.ora, dt.perc);
47     return 0;
48 }
```

Függvény határozott integráljának kiszámítása

Problémafelvetés és specifikáció

• Problémafelvetés

- Adott függvény határozott integrálját közelítsük egy beolvasott intervallumon a felhasználó által megadott számú részre osztva az intervallumot!

• Specifikáció

- A probléma inputja az integrálandó függvény neve, a , b valós számok, az intervallum végpontjai, n , a közelítéshez használandó részek száma.
- Az inputot a parancssorból vegye a program.
- Az output három valós szám: a megadott függvény értéke a és b pontokban, valamint a határozott integrál értéke.
- Ha az argumentumok száma nem megfelelő, akkor hibaüzenetet írunk ki.
- A függvény neve lehet az, hogy help, és ekkor csak tájékoztató szöveg jelenik meg a választható függvények nevééről.

Függvény határozott integráljának kiszámítása

Algoritmustervezés

- A fő algoritmusban:
 - Ellenőrizzük a parancssorban megadott argumentumok számát.
 - Megkeressük a függvény nevét a választható függvények közül.
 - Ha nincs meg, akkor tájékoztató és kész.
 - Ha help, akkor másik tájékoztató és kész.
 - Ha kevés a parancssorban megadott argumentumok száma, akkor hibaüzenet és kész.
 - Argumentumok konvertálása és a kiíratásban a megfelelő függvények meghívása és sikeres befejezés.



A -> operátor

- A megvalósítás során szükségünk lesz olyan hivatkozásokra, mint

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

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

- 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.
- A -> operátor segítségével tehát a fenti alakú hivatkozások

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

egyszerűbb alakban írhatóak.

C műveletek prioritása

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

Függvény határozott integráljának kiszámítása [1/5]

fgvint.c [1–24]

```
1 /* Közelítő integrálás a trapéz szabály segítségével.
2  * Az integrálandó függvény nevét a parancssorból kapjuk.
3  * Meg kell adnunk az intervallumot is és a finomságot is.
4  * Kérhető help is.
5  * 1998. Április 14. Dévényi Károly, devenyi@inf.u-szeged.hu
6  * 2020. Július 24. Gergely Tamás, gertom@inf.u-szeged.hu
7  */
8
9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h>
12 #include <math.h>
13 #include <errno.h>
14
15 typedef struct tablaelem_t {
16     char *nev;                                /* A függvény neve szövegesen */
17     double (*fuggvény)();                    /* A függvényre mutató pointer */
18 } tablaelem_t;
19
20 static double help();
21
22 static double expx();
23
24 static double sin2x();
```

Függvény határozott integráljának kiszámítása [2/5]

fgvint.c [26–42]

```
26 tablaelem_t tablazat[] = {
27     { "sin",    sin  },
28     { "tan",    tan  },
29     { "cos",    cos  },
30     { "expx",   expx  },
31     { "sin2x",  sin2x },
32     { "help",   help  },
33     { NULL,     NULL  }
34 };
35
36 static double expx(double x) {
37     return (exp(x) / x);
38 }
39
40 static double sin2x(double x) {
41     return sin(2.0 * x);
42 }
```

/ A táblázat végét jelzi */*



Függvény határozott integráljának kiszámítása [3/5]

fgvint.c [44–67]

```
44 double help(tablaelem_t *tp) {
45     printf("Kérem válasszon a következő függvények közül:");
46     for (; tp -> nev; ++tp) {
47         printf("%s", tp -> nev);
48     }
49     printf("\nMeg kell adnia az intervallumot és a finomságot.\n");
50     exit(EXIT_SUCCESS);
51 }
52                                     /* Az integrálandó függvény típusa */
53 typedef double (*fuggveny_t)(double x);
54
55 static double trapez(fuggveny_t f,                                     /* f(x)-t integráljuk az */
56                     double a, double b,                             /* a,b intervallumon */
57                     int n) {                                         /* n részre osztva az intervallumot */
58     /* Közelítő integrálás a trapéz szabály segítségével. */
59     const double h = (b - a) / n;
60     double area = 0.0;
61     int i;                                                           /* a ciklusváltozó */
62
63     for (i = 1; i < n; ++i) {                                       /* fgv. értékek összegzése */
64         area += (*f)(a + i * h);
65     }
66     return (area * h + ((*f)(a) + (*f)(b)) / 2.0 * h);
67 }
```

Függvény határozott integráljának kiszámítása [4/5]

fgvint.c [69–93]

```
69 int main(int argc, char **argv) {
70     tablaelem_t *tp;
71     double a, b;
72     int n;
73     char *kiiratas_formatuma = "\n%s(%)= %g\n%s(%)= %g\n%s integrálja= %g\n";
74
75     if (argc > 5) {
76         errno = E2BIG;
77         perror("A_hiba_az_hogy");
78         exit(EXIT_FAILURE);
79     }
80
81     for (tp = tablazat; /* a függvény nevét keressük */
82         (tp -> nev && argv[1] && strcmp(tp -> nev, argv[1]));
83         tp++)
84         ; /* üres ciklusmag */
85
86     if (tp -> nev == NULL) { /* nem találtuk meg */
87         printf("%sfüggvény_még_nincs_megvalósítva\n", argv[1]);
88         exit(EXIT_SUCCESS);
89     }
90
91     if (tp -> fuggveny == help) {
92         help(tablazat);
93     }
```

Függvény határozott integráljának kiszámítása [5/5]

fgvint.c [95–114]

```
95     if (argc < 5) {
96         perror("A_hiba_az,,hogy_kevés_az_argumentum");
97         exit(EXIT_FAILURE);
98     }
99
100     a = atof(argv[2]);           /* Konvertál sztring-ről double-re */
101     b = atof(argv[3]);
102     n = atoi(argv[4]);           /* Konvertál string-ről int-re */
103     printf(kiiratas_formatuma,  /* Pointer a formátum sztringre */
104         argv[1],                 /* Pointer a függvény nevére */
105         argv[2],                 /* Pointer az első számra */
106         (*tp -> fuggveny)(a),    /* A függvény értéke a-nál */
107         argv[1],                 /* Pointer a függvény nevére */
108         argv[3],                 /* Pointer a második számra */
109         (*tp -> fuggveny)(b),    /* A függvény értéke b-nél */
110         argv[1],                 /* Pointer a függvény nevére */
111         trapez(*tp -> fuggveny, a, b, n) /* A függvény integrálja */
112     );
113     exit(EXIT_SUCCESS);
114 }
```

Síkidomok területe és kerülete

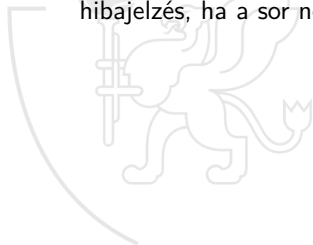
Problémafelvetés és specifikáció

- Problémafelvetés

- Határozzuk meg kör, téglalap és háromszög alakú síkidomok területét és kerületét.

- Specifikáció

- A probléma inputja több sor „ $\text{kor}(r)$ ”, „ $\text{teglalap}(a,b)$ ” vagy „ $\text{haromszog}(a,b,c)$ ” alakú szöveggel, ahol r , a , b és c valós számok. Az inputot a fájl vége (ctrl+D) zárja.
- Az output a megadott síkidom területe és kerülete két sorban, vagy egy hibajelzés, ha a sor nem megfelelő formátumú.



- Algoritmustervezés:

- Az inputot soronként olvassuk be, és ha megfelel valamelyik síkidom leírásának, akkor feltöltünk egy síkidom adattípust (`union`-ok és `struct`-ok kombinációja) a megfelelő adatokkal.
- Az említett adattípus tartalmazza a megfelelő terület- és kerületszámító függvények pointereit is.
- A fő algoritmusban csak az input adatokat olvassuk be, majd meghívjuk a síkidom terület és kerületszámító függvényeit, és kiírjuk a kiszámolt értékeket.



Síkidomok 2. verzió [1/5]

sikidomok-fgv-ptr.c [1–12]

```
1 /* Síkidomok területének és kerületének kiszámítása.
2  * 2018. Szeptember 19. Gergely Tamás, gertom@inf.u-szeged.hu
3  */
4
5 #include <math.h>
6 #include <stdio.h>
7
8 #define PUFFERMERET 128
9
10 enum sikidom_típus_t {
11     kor, haromszog, teglalap
12 };
```



Síkidomok 2. verzió [2/5]

sikidomok-fgv-ptr.c [14–35]

```
14 struct kor_tulajdonsagok_t {
15     double r;
16 };
17
18 struct haromszog_tulajdonsagok_t {
19     double a, b, c;
20 };
21
22 struct teglalap_tulajdonsagok_t {
23     double a, b;
24 };
25
26 struct sikidom {
27     enum sikidom_tipus_t tipus;
28     union {
29         struct kor_tulajdonsagok_t kor;
30         struct haromszog_tulajdonsagok_t haromszog;
31         struct teglalap_tulajdonsagok_t teglalap;
32     };
33     double (*kerulet)(struct sikidom);
34     double (*terulet)(struct sikidom);
35 };
```

Síkidomok 2. verzió [3/5]

sikidomok-fgv-ptr.c [37–62]

```
37 double kor_kerulet(struct sikidom si) {
38     return (si.tipus == kor) ? 2.0 * M_PI * si.kor.r : NAN;
39 }
40
41 double kor_terulet(struct sikidom si) {
42     return (si.tipus == kor) ? M_PI * si.kor.r * si.kor.r : NAN;
43 }
44
45 double haromszog_kerulet(struct sikidom si) {
46     return (si.tipus == haromszog) ?
47         si.haromszog.a + si.haromszog.b + si.haromszog.c : NAN;
48 }
49
50 double haromszog_terulet(struct sikidom si) {
51     double s = (si.haromszog.a + si.haromszog.b + si.haromszog.c) / 2.0;
52     return (si.tipus == haromszog) ?
53         sqrt(s * (s-si.haromszog.a) * (s-si.haromszog.b) * (s-si.haromszog.c)) : NAN;
54 }
55
56 double teglalap_kerulet(struct sikidom si) {
57     return (si.tipus == teglalap) ? 2.0 * (si.teglalap.a + si.teglalap.b) : NAN;
58 }
59
60 double teglalap_terulet(struct sikidom si) {
61     return (si.tipus == teglalap) ? si.teglalap.a * si.teglalap.b : NAN;
62 }
```


Síkidomok 2. verzió [4/5]

sikidomok-fgv-ptr.c [64–92]

```
64 struct sikidom uj_kor(double r) {
65     struct sikidom s;
66     s.tipus      = kor;
67     s.kor.r      = r;
68     s.kerulet    = kor_kerulet;
69     s.terulet    = kor_terulet;
70     return s;
71 }
72
73 struct sikidom uj_teglalap(double a, double b) {
74     struct sikidom s;
75     s.tipus      = teglalap;
76     s.teglalap.a = a;
77     s.teglalap.b = b;
78     s.kerulet    = teglalap_kerulet;
79     s.terulet    = teglalap_terulet;
80     return s;
81 }
82
83 struct sikidom uj_haromszog(double a, double b, double c) {
84     struct sikidom s;
85     s.tipus      = haromszog;
86     s.haromszog.a = a;
87     s.haromszog.b = b;
88     s.haromszog.c = c;
89     s.kerulet    = haromszog_kerulet;
90     s.terulet    = haromszog_terulet;
91     return s;
92 }
```

Síkidomok 2. verzió [5/5]

sikidomok-fgv-ptr.c [94–113]

```
94 char puffer[PUFFERMERET];
95
96 int main() {
97     double a,b,c;
98     struct sikidom s;
99     while(fgets(puffer, PUFFERMERET, stdin)) {
100         if(sscanf(puffer, "kor(%lf)", &a) == 1) {
101             s = uj_kor(a);
102         } else if(sscanf(puffer, "teglalap(%lf,%lf)", &a, &b) == 2) {
103             s = uj_teglalap(a, b);
104         } else if(sscanf(puffer, "haromszog(%lf,%lf,%lf)", &a, &b, &c) == 3) {
105             s = uj_haromszog(a, b, c);
106         } else {
107             printf("Ismeretlen formatumu sor!\n");
108             continue;
109         }
110         printf("T=%lf\nK=%lf\n", s.terulet(s), s.kerulet(s));
111     }
112     return 0;
113 }
```

- 1 **Bemutakozás**
 - Kurzus információk
 - A SZTE és az informatikai képzés
- 2 **Linux**
 - Alapfogalmak
 - Linux parancsok
 - Linux shell
 - Felhasználók
 - Hálózat
- 3 **Gyors C áttekintés**
 - Bevezető
 - Pénzváltás (1. verzió)
 - Pénzváltás (2. verzió)
 - Röppálya számítás
 - Röppálya szimuláció
 - Az év napja
 - Csúszóátlag adott elemszámmra
 - Csúszóátlag parancssorból
 - Basename standard inputról
 - Basename parancssorból
 - Tér legtávolabbi pontjai
 - A nappalis gyakorlat értékelése

- 4 **Alapok**
 - Alapfogalmak
 - A programozás fázisai
 - Algoritmus vezérlése
 - A C nyelvű program
 - Szintaxis
 - A C nyelv elemi adattípusai
 - A C nyelv utasításai
- 5 **Vezérlési szerkezetek**
 - Bevezetés
 - Szekvenciális vezérlés
 - Függvények
 - Szelekciós vezérlések
 - Ismétléses vezérlések 1.
 - Eljárásvezérlés
 - Ismétléses vezérlések 2.
- 6 **Folyamatábra és struktúradiagram**
- 7 **Adatszerkezetek**
 - Az adatkezelés szintjei
 - Elemi adattípusok
 - Pointer adattípus
 - Tömb adattípus

- 8 **Sztringek**
 - Pointerek és tömbök C-ben
 - Rekord adattípus
 - Függvény pointer
 - **Halmaz adattípus**
 - Flexibilis tömbök
 - Láncolt listák
 - Típusokról C-ben
- 9 **IO**
 - Alapok
 - Adatállományok
- 10 **C fordítás**
 - A fordítás folyamata
 - A preprocesszor
 - A C fordító
 - Assembler
 - Linker és modulok
- 11 **Gyakorlati kérdések**
 - Memóriahasználat
 - Gyakori C hibák
 - where.c felboncolva

- A programozásban számtalan formában előfordulhatnak halmazok és halmazokkal végzett műveletek.
- Legyen U egy egész típus, amit univerzumnak nevezünk. Tekintsük a $P(U)$ érték-halmazt, vagyis az U univerzum részhalmazainak halmazát, ez lesz a $Halmaz(U)$ új adattípus érték-halmaza.



Halmaz absztrakt adattípus

Műveletek [1/2]

- $\text{Üresít}(\leftarrow H:\text{Halmaz});$
 - A művelet végrehajtása után a H változó értéke az üres halmaz lesz.
- $\text{Bővít}(\leftrightarrow H:\text{Halmaz}; \rightarrow x:U);$
 - A művelet a H változó értékéhez hozzáveszi az x elemet.
- $\text{Töröl}(\leftrightarrow H:\text{Halmaz}; \rightarrow x:U);$
 - A művelet a H változó értékéből törli az x elemet.
- $\text{Eleme}(\rightarrow x:U; \rightarrow H:\text{Halmaz}): \text{bool};$
 - A függvényművelet akkor és csak akkor ad igaz értéket, ha $x \in H$ halmaznak.
- $\text{Egyesítés}(\rightarrow H_1:\text{Halmaz}; \rightarrow H_2:\text{Halmaz}; \leftarrow H:\text{Halmaz});$
 - A művelet eredményeként a H változó értéke a $H_1 \cup H_2$ lesz.
- $\text{Metszet}(\rightarrow H_1:\text{Halmaz}; \rightarrow H_2:\text{Halmaz}; \leftarrow H:\text{Halmaz});$
 - A művelet eredményeként a H változó értéke a $H_1 \cap H_2$ lesz.

Halmaz absztrakt adattípus

Műveletek [2/2]

- Különbség($\rightarrow H_1:\text{Halmaz}; \rightarrow H_2:\text{Halmaz}; \leftarrow H:\text{Halmaz}$);
 - A művelet eredményeként a H változó azokat az $x \in H_1$ elemeket tartalmazza, melyekre $x \notin H_2$.
- Egyenlő($\rightarrow H_1:\text{Halmaz}; \rightarrow H_2:\text{Halmaz}$):bool;
 - Az egyenlőség relációs művelet.
- Rész($\rightarrow H_1:\text{Halmaz}; \rightarrow H_2:\text{Halmaz}$):bool;
 - Akkor és csak akkor ad igaz értéket, ha a $H_1 \subseteq H_2$.
- Ürese($\rightarrow H:\text{Halmaz}$):bool;
 - Akkor és csak akkor ad igaz értéket, ha $H = \emptyset$.
- Értékadás($\leftarrow H_1:\text{Halmaz}; \rightarrow H_2:\text{Halmaz}$);
 - A művelet hatására a H_1 változó felveszi a H_2 értékét.

- A C nyelvben a halmaznak nincs nyelvi megvalósítása.
- A halmazok reprezentálásához induljunk ki abból, hogy tetszőleges U univerzum esetén az U részhalmazai megadhatók karakterisztikus függvényükkel. Ha H az U részhalmaza, akkor karakterisztikus függvénye:

$$k_H : U \rightarrow \{0, 1\}$$

$$k_H(x) = \begin{cases} 1 & \Leftrightarrow x \in H \\ 0 & \Leftrightarrow x \notin H \end{cases}$$

- Vagyis minden részhalmazhoz egyértelműen hozzárendelhető egy olyan függvény, ami a részhalmazban lévő elemekre 1-et, az univerzum többi elemére pedig 0-t ad.

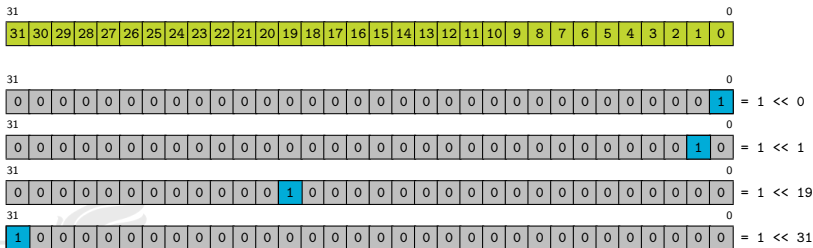
- Ha U egész típus, a karakterisztikus függvényeket meg tudnánk valósítani a `bool[U]` típussal, de ez nem hatékony megoldás, mert U minden eleméhez a logikai `bool` típus megvalósításától függően legalább egy (de inkább 4) byte szükséges.
- Ezzel szemben az 1 bit/elem hatékonyságot el is tudjuk érni a bitműveletek segítségével. A módszer lényege, hogy például egy `int` típusú változóban (32 bites rendszeren) 32 bitet tárolhatunk, azaz egy 32 elemű kis halmaz reprezentálására ideális.
 - Egész egyszerűen minden bitnek feleltessünk meg egy értéket, és viszont. Ez nagyon sokféleképpen (32!) megtehető, például:

31																																						0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31							

vagy

31																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																						
----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

- Az egyes biteket a bitműveletekkel érhetjük el.



- A bit törlése, beállítása és lekérdezése az $\&$, $|$ és \sim műveletek segítségével történik.

- Nagyobb halmazt összerakhatunk kis halmazokból a következőképpen:

Logikailag

$K-1$...	0
$2K-1$...	K
$(i+1)K-1$...	x
	...	iK
$H_{max}-1$...	$(M-1)K$

Fizikailag

tömbindex

[0]	$K-1$...	0
[1]	$K-1$...	0
$[i=(x/K)]$	$K-1$...	$x \% K$
		...	0
[M-1]	$K-1$...	0

- Minden x természetes szám egyértelműen meghatározott az $(x/K, x\%K)$ számpárral.
- Tehát x értékéből egyértelműen előállítható, hogy a halmazt reprezentáló H nagy tömb melyik kishalmazba milyen pozíción tartalmazza az x -hez tartozó bitet.
- x akkor és csak akkor eleme a H változó által reprezentált halmaznak, ha teljesül, hogy a $H[x/K]$ kishalmaznak eleme az $(x\%K)$, azaz a megfelelő kishalmaz megfelelő bitje 1.
- Az univerzumot a tömb és a használt adattípus mérete együtt határozzák meg.

Halmaz megvalósítás [1/4]

halmaz_t.h [1–26]

```
1 #ifndef HALMAZ_H
2 #define HALMAZ_H
3
4 #include <stdbool.h>
5
6 #define K (8*sizeof(KisHalmaz))
7 #define M ???
8 #define H_MAX (M*K)
9
10 typedef unsigned long int halmazelem;
11 typedef halmazelem KisHalmaz;
12 typedef KisHalmaz Halmaz[M];
13
14 void Uresit(Halmaz);
15 void Bovit(Halmaz, halmazelem);
16 void Torol(Halmaz, halmazelem);
17 bool Eleme(Halmaz, halmazelem);
18 void Egyesites(Halmaz H1, Halmaz H2, Halmaz H);
19 void Metszet(Halmaz H1, Halmaz H2, Halmaz H);
20 void Kulonbseg(Halmaz H1, Halmaz H2, Halmaz H);
21 bool Egyenlo(Halmaz H1, Halmaz H2);
22 bool Resz(Halmaz H1, Halmaz H2);
23 bool Urese(Halmaz H);
24 void Ertekadas(Halmaz H1, Halmaz H2);
25
26 #endif // HALMAZ_H
```

Halmaz megvalósítás [2/4]

halmaz_t.c [1-21]

```
1#include "halmaz_t.h"
2
3void Uresit(Halmaz H) {
4    long int i;
5    for(i = 0; i < M; ++i)
6        H[i] = 0;
7}
8
9void Bovit(Halmaz H, halmazelem x) {
10    if(x < H_MAX)
11        H[x / K] |= (1 << (x % K));
12}
13
14void Torol(Halmaz H, halmazelem x) {
15    if(x < H_MAX)
16        H[x / K] &= ~(1 << (x % K));
17}
18
19bool Eleme(halmazelem x, Halmaz H) {
20    return (x < H_MAX) && (H[x / K] & (1 << (x % K)));
21}
```

Halmaz megvalósítás [3/4]

halmaz_t.c [23–45]

```
23 void Egyesites(Halmaz H1, Halmaz H2, Halmaz H) {
24     long int i;
25     for(i = 0; i < M; ++i)
26         H[i] = H1[i] | H2[i];
27 }
28
29 void Metszet(Halmaz H1, Halmaz H2, Halmaz H) {
30     long int i;
31     for(i = 0; i < M; ++i)
32         H[i] = H1[i] & H2[i];
33 }
34
35 void Kulonbseg(Halmaz H1, Halmaz H2, Halmaz H) {
36     long int i;
37     for(i = 0; i < M; ++i)
38         H[i] = H1[i] & ~(H2[i]);
39 }
40
41 bool Egyenlo(Halmaz H1, Halmaz H2) {
42     long int i;
43     for(i = 0; (i < M) && (H1[i] == H2[i]); ++i);
44     return i == M;
45 }
```

Halmaz megvalósítás [4/4]

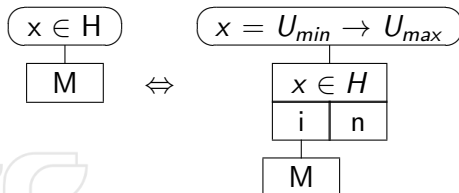
halmaz_t.c [47–63]

```
47 bool Resz(Halmaz H1, Halmaz H2) {
48     long int i;
49     for(i = 0; (i < M) && !(H1[i] & ~(H2[i])); ++i);
50     return i == M;
51 }
52
53 bool Urese(Halmaz H) {
54     long int i;
55     for(i = 0; (i < M) && !(H[i]); ++i);
56     return i == M;
57 }
58
59 void Ertekadas(Halmaz H1, Halmaz H2) {
60     long int i;
61     for(i = 0; i < M; ++i)
62         H1[i] = H2[i];
63 }
```



Diszkrét ismétléses vezérlés

- Azáltal, hogy az univerzum egész típus, lehetővé válik a diszkrét ismétléses vezérlés megfogalmazása ezekre a halmazokra.
- Ha U_{min} és U_{max} az univerzum minimális illetve maximális eleme, akkor a diszkrét ismétléses vezérlés megvalósítható az alábbi módon:



```
for (x = Umin; x <= Umax; ++x) {  
    if (Elem(x, H)) {  
        M;  
    }  
}
```


Klikkek meghatározása

Problémafelvetés és specifikáció

- Problémafelvetés

- Egy közösségben az emberek közötti barátsági kapcsolat alapján meg kell határoznunk a klikkeket!

- Specifikáció

- Legyen egy előre adott N az emberek száma, akiket jelöljünk az $1, \dots, N$ számokkal.
- A program inputja (i, j) számpárok sorozata ($1 \leq i, j \leq N$), ami az i . és j . sorszámú ember barátságát jelenti. A sorozat (és az input) végét a $(0, 0)$ számpár jelenti.
- Az output a baráti csoportok tagjainak felsorolása.

- A matematika nyelvén:
 - Tegyük fel, hogy a barátság reflexív, szimmetrikus és tranzitív reláció. Meg kell határozni az R relációt tartalmazó legszűkebb ekvivalencia reláció szerinti osztályozást.
 - Induljunk ki abból, hogy minden személy csak saját magával van barátságban (reflexív); tehát képezzük az i egyelemű halmazokat.
 - Ha az input számpárokat valameddig feldolgozva meghatároztuk az osztályozást, a következő (i, j) számpárt beolvasva össze kell vonni azt a két részhalmazzal, amelybe i illetve j tartozik, hisz mindenki, aki i -vel barátságban van, az barátságban van j minden barátjával is (tranzitív), és ez fordítva is igaz (szimmetrikus).

Klikkek meghatározása absztrakt Halmaz típussal [1/2]

klikk-abstract.c [1–28]

```
1 /* A beolvasott R relációt tartalmazó legszűkebb ekvivalencia
2    reláció szerinti osztályozást határozzuk meg.
3    Vázlat, nem fordítható C program.
4    2006. Augusztus 14. Gergely Tamás, gertom@inf.u-szeged.hu
5 */
6 #include <stdio.h>
7 #define N 10                                /* maximális elemszám */
8 typedef Halmaz(U) Halmaz;                  /* EZ ÍGY NEM C !!! */
9
10 int main() {
11     Halmaz H[N];
12     unsigned short i,j, ti,tj;
13     for(i = 1; i <= N; ++i) {                /* inicializálás */
14         Uresit(H[i-1]); Bovit(H[i-1], i);
15     }
16     printf("Kérem a relációban lévő számpárokat!\n");
17     scanf("%hd%hd%[^\\n]", &i, &j);
18     getchar();
19     while(i != 0) {
20         for(ti=0; !Eleme(i, H[ti]); ti++); /* amelyik H[ti] halmazban van i ? */
21         for(tj=0; !Eleme(j, H[tj]); tj++); /* amelyik H[tj] halmazban van j ? */
22         if(ti != tj) {                        /* H[ti] és H[tj] összevonása */
23             Egyesites(H[ti], H[tj], H[ti]);
24             Uresit( H[tj] );
25         }
26         scanf("%hd%hd%[^\\n]", &i, &j);
27         getchar();
28     }
```

Klikkek meghatározása absztrakt Halmaz típussal [2/2]

klikk-abstract.c [30–40]

```
30 printf("Az osztályok:\n");
31 for (i = 1; i <= N; i++) {
32     if (! Urese(H[i - 1])) {
33         for (j = 1; j <= N; j++) {
34             if (Eleme(j, H[i - 1]))
35                 printf("%4hd", j);
36         }
37         putchar('\n');
38     }
39 }
40 }
```

/ az osztályok kiírása */*



Klikkek meghatározása [1/2]

klikk.c [1–19]

```
1 /* A beolvasott R relációt tartalmazó legszűkebb ekvivalencia
2 * reláció szerinti osztályozást határozzuk meg.
3 * 1997. December 6. Dévényi Károly, devenyi@inf.u-szeged.hu
4 */
5
6 #include <stdio.h>
7
8 #define N 10 /* maximális elemszám */
9
10 typedef unsigned short int univerzum_t; /* az univerzum */
11 typedef long int halmaz_t; /* N kicsi, ezért elegendő egy long int */
12
13 int main(int argc, char *argv[]) {
14     halmaz_t h[N];
15     univerzum_t i, j, ti, tj;
16
17     for (i = 1; i <= N; ++i) { /* inicializálás */
18         h[i - 1] = 11 << i;
19     }
```



Klikkek meghatározása [2/2]

klikk.c [21–46]

```
21 printf("Kérem a relációban lévő számpárokat!\n");
22 scanf("%hd%hd%*[\n]", &i, &j); getchar();
23 while (i != 0) {
24     /* az i-t tartalmazó halmaz ti indexének keresése */
25     for (ti = 0; ((11 << i) & h[ti]) == 0; ++ti);
26     /* a j-t tartalmazó halmaz tj indexének keresése */
27     for (tj = 0; ((11 << j) & h[tj]) == 0; ++tj);
28     if (ti != tj) { /* h[ti] és h[tj] összevonása */
29         h[ti] |= h[tj];
30         h[tj] = 0;
31     }
32     scanf("%hd%hd%*[\n]", &i, &j); getchar();
33 }
34
35 printf("Az osztályok:\n");
36 for (i = 1; i <= N; ++i) { /* az osztályok kiírása */
37     if (h[i - 1] != 0) {
38         for (j = 1; j <= N; ++j) {
39             if (((11 << j) & h[i - 1]) != 0)
40                 printf("%4d,", j);
41         }
42         putchar('\n');
43     }
44 }
45 return 0;
46 }
```

Prímszámok meghatározása

Problémafelvetés és specifikáció

- Problémafelvetés
 - Határozzuk meg az adott N természetes számnál nem nagyobb prímszámokat.
- Specifikáció
 - Legyen N egy előre adott pozitív egész szám.
 - A programnak nincs bemenete.
 - A kimenet a $[0, N)$ intervallumba eső prímszámok sorozata.



Prímszámok meghatározása

Algoritmustervezés

- Első ötletünk egy „favágó” algoritmus lehet: minden számról külön-külön eldöntjük, hogy prím-e vagy sem.
- Szerkezeti ábra:

Prímszámok

$d = 2 \rightarrow N - 1$

prím-e(p)	
i	n

Ki(p)

bool prím-e(int n)

$d = 2 \rightarrow \lfloor \sqrt{n} \rfloor$

return $n > 1$

n osztható d -vel	
i	n

return false

Prímszámok meghatározása [1/2]

prim-nem-hatekony.c [1–18]

```
1 /* Határozzuk meg az adott N természetes számnál nem nagyobb
2  * prímszámokat.
3  * 2014. Január 24. Gergely Tamás, gertom@inf.u-szeged.hu
4  */
5
6 #include <stdio.h>
7
8 #define N 16777216LL
9
10 int prim_e(long long int n) {
11     long long int d;
12     for (d = 2; d * d <= n; ++d) {
13         if (n % d == 0) {
14             return 0;
15         }
16     }
17     return 1 < n;
18 }
```



Prímszámok meghatározása [2/2]

prim-nem-hatekony.c [20–34]

```
20 int main() {
21     long long int p, j = 0;
22     printf("A prímszámok %-lld-ig: \n%9lld", N, 2LL);
23     for (p = 3; p < N; p += 2) {
24         if (prim_e(p)) {
25             printf("%9lld", p);
26             if (++j == 8) {
27                 j = 0;
28                 putchar('\n');
29             }
30         }
31     }
32     putchar('\n');
33     return 0;
34 }
```



Prímszámok meghatározása

Algoritmustervezés

- A „minden számról külön-külön eldöntjük, hogy prím-e vagy sem” algoritmus nem igazán hatékony
 - Összehasonlíthatjuk például a `prim` és `prim-nem-hatekony` futási idejét:

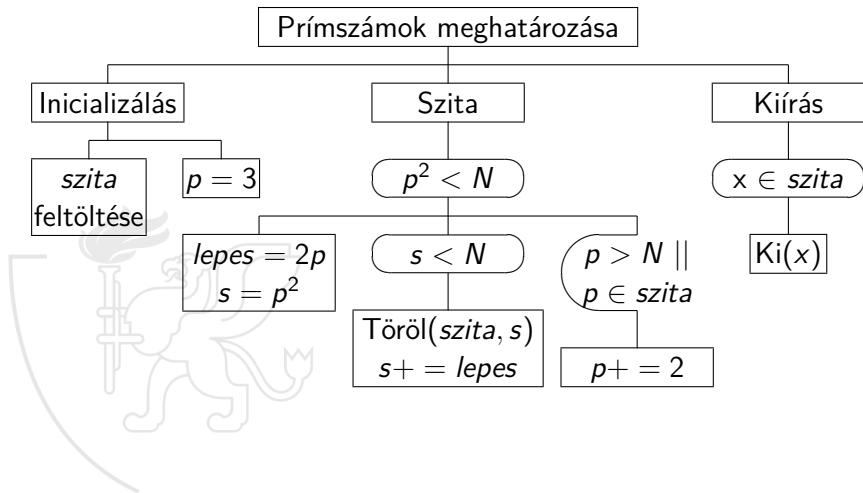
```
$ /usr/bin/time -f '%E (%Us)' ./prim >/dev/null  
$ /usr/bin/time -f '%E (%Us)' ./prim-nem-hatekony >/dev/null
```

- Ezért a jól ismert Erathosztenészi szita algoritmust valósítjuk meg:
 - A halmazt kezdetben feltöltjük az egynél nagyobb páratlan számokkal.
 - Megkeressük a halmaz még nem feldolgozott legkisebb elemét (ez prímszám lesz) és töröljük a többszöröseit.
 - Az előző pontot addig ismételjük, míg el nem érjük az N gyökét.
 - Az eredményhalmaz csak a prímszámokat fogja tartalmazni.

Prímszámok meghatározása

Algoritmustervezés – Szerkezeti ábra

- Szerkezeti ábra:



Prímszámok meghatározása [1/3]

prim.c [1–14]

```
1 /* Határozzuk meg az adott N természetes számnál nem nagyobb
2  * prímszámokat.
3  * 1997. December 6. Dévényi Károly, devenyi@inf.u-szeged.hu
4  * 2006. Augusztus 15. Gergely Tamás, gertom@inf.u-szeged.hu
5  */
6
7 #include <stdio.h>
8
9 #define K (8 * sizeof(kishalmaz_t))
10 #define M 262144LL
11 #define N (M * K)
12
13 typedef long long int kishalmaz_t;
14 typedef kishalmaz_t halmaz_t[M];
```



Prímszámok meghatározása [2/3]

prim.c [16–42]

```
16 int main() {
17     halmaz_t szita;
18     kishalmaz_t kicsi;                                /* a szita inicializálásához */
19     long long int p, s, lepes, i, j;
20
21     kicsi = 0;                                         /* a szita inicializálása */
22     for (i = 0; i <= ((K - 1) / 2); ++i) {
23         kicsi |= (1LL << (2 * i + 1));
24     }
25     for (i = 0; i < M; ++i) {
26         szita[i] = kicsi;
27     }
28     szita[0] &= ~2LL; /* 2LL == (1LL << 1) */
29     szita[0] |= 4LL; /* 4LL == (1LL << 2) */
30
31     p = 3;                                             /* az első szítálandó prím */
32     while (p * p < N) {                                /* P többszöröseinek kiszitálása */
33         lepes = 2 * p;                                /* lépésköz = 2*p */
34         s = p * p;                                     /* s az első többszörös */
35         while (s < N) {
36             szita[s / K] &= ~(1LL << (s % K));
37             s += lepes;
38         }
39         do {                                           /* a következő prím keresése */
40             p += 2;
41         } while ((p < N) && !(szita[p / K] & (1LL << (p % K))));
42     }
```

Prímszámok meghatározása [3/3]

prim.c [44–57]

```
44     j = 0;                                     /* a prímszámok kiírása képernyőre */
45     printf("A prímszámok %lld-ig:\n", N);
46     for (p = 2; p < N; ++p) {
47         if (szita[p / K] & (1LL << (p % K))) {
48             printf("%9lld", p);
49             if (++j == 8) {
50                 j = 0;
51                 putchar('\n');
52             }
53         }
54     }
55     putchar('\n');
56     return 0;
57 }
```



- 1 **Bemutakozás**
 - Kurzus információk
 - A SZTE és az informatikai képzés
- 2 **Linux**
 - Alapfogalmak
 - Linux parancsok
 - Linux shell
 - Felhasználók
 - Hálózat
- 3 **Gyors C áttekintés**
 - Bevezető
 - Pénzváltás (1. verzió)
 - Pénzváltás (2. verzió)
 - Röppálya számítás
 - Röppálya szimuláció
 - Az év napja
 - Csúszóátlag adott elemszámmra
 - Csúszóátlag parancssorból
 - Basename standard inputról
 - Basename parancssorból
 - Tér legtávolabbi pontjai
 - A nappalis gyakorlat értékelése

- 4 **Alapok**
 - Alapfogalmak
 - A programozás fázisai
 - Algoritmus vezérlése
 - A C nyelvű program
 - Szintaxis
 - A C nyelv elemi adattípusai
 - A C nyelv utasításai
- 5 **Vezérlési szerkezetek**
 - Bevezetés
 - Szekvenciális vezérlés
 - Függvények
 - Szelekciós vezérlések
 - Ismétléses vezérlések 1.
 - Eljárásvezérlés
 - Ismétléses vezérlések 2.
- 6 **Folyamatábra és struktúradiagram**
- 7 **Adatszerkezetek**
 - Az adatkezelés szintjei
 - Elemi adattípusok
 - Pointer adattípus
 - Tömb adattípus

- Sztringek
 - Pointerek és tömbök C-ben
 - Rekord adattípus
 - Függvény pointer
 - Halmaz adattípus
 - **Flexibilis tömbök**
 - Láncolt listák
 - Típusokról C-ben
- 8 **IO**
 - Alapok
 - Adatállományok
 - 9 **C fordítás**
 - A fordítás folyamata
 - A preprocesszor
 - A C fordító
 - Assembler
 - Linker és modulok
 - 10 **Gyakorlati kérdések**
 - Memóriahasználat
 - Gyakori C hibák
 - where.c felboncolva

Változó-hosszúságú tömb C-ben

- Az eddig megismert C tömbökkel az a baj, hogy a méretüket fordítási időben meg kell adni.
 - A feldolgozott adatok mennyisége, mérete viszont nem mindig ismert előre, így előfordulhat, hogy a tömb számára kevés helyet foglaltunk, de az is, hogy feleslegesen sokat.
- A **C99** szabvány már ismeri a változó-hosszúságú tömb fogalmát, amikor is a tömb deklarációjában méretként egy futásidőben kiértékelhető egész kifejezést is megadhatunk.
 - Ez viszont a memória nem pont erre tervezett *verem* részében fog helyet foglalni, és emiatt korlátozottan használható.
 - Ráadásul a **C11** szabvány bizonyos implementációkra felmentést ad az ilyen tömbök kezelése alól.

```
void func(int n) {  
    int tomb[n];  
    ...  
}
```

- A megoldás: tömb helyett pointert deklarálunk, és ha tudjuk a kívánt méretet, memóriát már a megfelelő számú elemnek foglalunk.
 - Vagyis dinamikusan foglalunk helyet a tömbünk elemeinek, azaz dinamikus tömböt használunk. Mivel a pointert tömbként kezelhetjük, a program kódjában ez semmilyen más változást nem eredményez.

```
int tomb[MAX];  
...  
n = ...  
  
...  
for(i=0; i<n; ++i) {  
    tomb[i]=i;  
}  
  
...
```

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

Egyszerű flexibilis tömb C-ben

- A hatékonyabb memóriakezelés ellenére a dinamikus tömb méretét még így is előre, legkésőbb a tömb létrehozásakor ismernünk kell.
- De erre is van megoldás: a C nyelvben a

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

függvény a ptr által mutatott (foglalt) területet méretezi át.

- Ennek segítségével meg tudunk valósítani egy egyszerű flexibilis tömböt, aminek a méretét nem kell előre megadnunk:

```
int *tomb = NULL, int n = 0;  
...  
while(...) {  
    tomb = realloc(tomb, (++n)*sizeof(int));  
    ...  
}  
free(tomb);  
...
```

Flexibilis tömb

Absztrakt adattípus – Értékkészlet

- Algoritmusok tervezésekor gyakran előfordul, hogy tömbökkel kell dolgozni, de az adatok darabszáma csak az algoritmus futása közben válik ismertté, esetleg a futás során még változhat is. Ez utóbbi esetet a megismert tömb adattípus nem tudja kezelni.
- Egy ilyen típus értékkészlete $\bigcup_{n=1}^{\infty} T_n$, ahol T_n az E elemtípusból és $I_n = \{0 \dots n - 1\}$ indextípusból képzett tömb típus.
- Ha az ilyen sorozatokon a következő műveleteket értelmezzük, akkor egy (absztrakt) adattípushoz jutunk, amit *Flexibilis tömb* típusnak nevezünk.
- Jelöljük ezt a *Flexibilis tömb* típust a továbbiakban FT -vel, és legyen $I = \{0 \dots \infty\}$.

- $\text{Kiolvas}(\rightarrow A:FT; \rightarrow i:I; \leftarrow x:E)$
 - Az A sorozat i . komponensének kiolvasása x változóba ha A legalább $i + 1$ elemet tartalmaz.
- $\text{Módosít}(\leftrightarrow A:FT; \rightarrow i:I; \rightarrow y:E)$
 - Az A sorozat i . komponensének módosítása y értékre, ha A legalább $i + 1$ elemet tartalmaz.
- $\text{Felső}(\rightarrow A:FT): I$
 - Az A elemszámának lekérdezése.
- $\text{Értékadás}(\leftarrow A:FT; \rightarrow X:FT)$
 - Értékadó művelet. Az A változó felveszi az X , FT típusú kifejezés értékét.

Flexibilis tömb

Absztrakt adattípus – Műveletek

- $\text{Létesít}(\leftrightarrow A:FT; \rightarrow n:I)$
 - n elemű flexibilis tömböt létesít.
- $\text{Meggzűntet}(\leftrightarrow A:FT\text{ömb})$
 - Törli az A flexibilis tömbhöz foglalt memóriát.
- $\text{Növel}(\leftrightarrow A:FT; \rightarrow d:I)$
 - Az A felső határát a d értékkel növeli.
- $\text{Csökkent}(\leftrightarrow A:FT; \rightarrow d:I)$
 - Az A felső határát a d értékkel csökkenti.

Flexibilis tömb

Virtuális C adattípus

- Egyszerűbb flexibilis tömb típust az alábbi módon deklarálhatunk:

```
typedef struct FT {  
    E *tomb;                                /* tömb */  
    I meret;  /* a tömb aktuális mérete */  
} FT;
```

vagyis a tömböt magát tulajdonképpen egy pointerrel valósítjuk meg. A struktúra azért kell, mert a flexibilis tömbhöz hozzátartozik a mérete is.



Flexibilis tömb egyszerű megvalósítás [1/3]

ftomb_simple_t.h [1-20]

```
1 #ifndef FTOMB_SIMPLE_H
2 #define FTOMB_SIMPLE_H
3
4 typedef ???          elem_t;          /* a tömb elemtípusa */
5 typedef unsigned int index_t;
6 typedef struct flex_tomb_t {
7     elem_t *tomb;                      /* tömb */
8     index_t hatar;                     /* aktuális indexhatár */
9 } flex_tomb_t;
10
11 /* A műveletek deklarációja: */
12 void kiolvas(flex_tomb_t a, index_t i, elem_t *x);
13 void modosit(flex_tomb_t a, index_t i, elem_t x);
14 index_t felso(flex_tomb_t a);
15 void letesit(flex_tomb_t *a, index_t n);
16 void megszuntet(flex_tomb_t *a);
17 void novel(flex_tomb_t *a, index_t d);
18 void csokkent(flex_tomb_t *a, index_t d);
19
20 #endif // FTOMB_SIMPLE_H
```


Flexibilis tömb egyszerű megvalósítás [2/3]

ftomb_simple_t.c [1-18]

```
1#include <stdlib.h>
2#include "ftomb_simple_t.h"
3
4void kiolvas(flex_tomb_t a, index_t i, elem_t *x) {
5    if (i < a.hatar) {
6        *x = a.tomb[i];
7    }
8}
9
10void modosit(flex_tomb_t a, index_t i, elem_t x) {
11    if (i < a.hatar) {
12        a.tomb[i] = x;
13    }
14}
15
16index_t felso(flex_tomb_t a) {
17    return a.hatar;
18}
```



Flexibilis tömb egyszerű megvalósítás [3/3]

ftomb_simple_t.c [20–37]

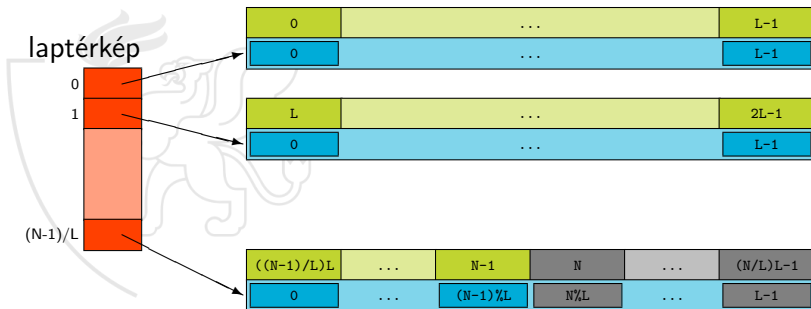
```
20 void letesit(flex_tomb_t *a, index_t n) {
21     a->hatar = n;
22     a->tomb = (elem_t*) ((n) ? malloc(n * sizeof(elem_t)) : NULL);
23 }
24
25 void megszuntet(flex_tomb_t *a) {
26     free(a->tomb);           /* A free() függvény jól kezeli a NULL értéket */
27     a->tomb = NULL;
28     a->hatar = 0;
29 }
30
31 void novel(flex_tomb_t *a, index_t d) {
32     a->tomb = (elem_t*)realloc(a->tomb, (a->hatar += d) * sizeof(elem_t));
33 }
34
35 void csokkent(flex_tomb_t *a, index_t d) {
36     a->tomb = (elem_t*)realloc(a->tomb, (a->hatar -= d) * sizeof(elem_t));
37 }
```



Flexibilis tömb

Virtuális C adattípus

- Az előző megvalósítás sem használható viszont, ha a memória túlságosan „széttöredezett”, azaz nincsenek benne nagy egybefüggő részek.
- Szerencsére a halmazok megvalósításánál már látott elvet újrahasznosító megvalósítás ezt a problémát is kiküszöböli (ha nem is teljesen).



Flexibilis tömb megvalósítás [1/4]

ftomb_t.h [1–24]

```
1 #ifndef FTOMB_H
2 #define FTOMB_H
3
4 #define L ???                                /* lapméret */
5
6 typedef ???          elem_t;                /* a tömb elemtípusa */
7 typedef unsigned int index_t;
8 typedef elem_t *lap_t;
9 typedef lap_t *lapterkep_t;
10 typedef struct flex_tomb_t {
11     lapterkep_t lt;                          /* laptérkép */
12     index_t     hatar;                      /* aktuális indexhatár */
13 } flex_tomb_t;
14
15 /* A műveletek deklarációja: */
16 void kiolvas(flex_tomb_t a, index_t i, elem_t *x);
17 void modosit(flex_tomb_t a, index_t i, elem_t x);
18 index_t felso(flex_tomb_t a);
19 void letesit(flex_tomb_t *a, index_t n);
20 void megszuntet(flex_tomb_t *a);
21 void novel(flex_tomb_t *a, index_t d);
22 void csokkent(flex_tomb_t *a, index_t d);
23
24 #endif // FTOMB_H
```

Flexibilis tömb megvalósítás [2/4]

ftomb_t.c [1–18]

```
1#include <stdlib.h>
2#include "ftomb_t.h"
3
4void kiolvas(flex_tomb_t a, index_t i, elem_t *x) {
5    if (i < a.hatar) {
6        *x = a.lt[i / L][i % L];
7    }
8}
9
10void modosit(flex_tomb_t a, index_t i, elem_t x) {
11    if (i < a.hatar) {
12        a.lt[i / L][i % L] = x;
13    }
14}
15
16index_t felso(flex_tomb_t a) {
17    return a.hatar;
18}
```



Flexibilis tömb megvalósítás [3/4]

ftomb_t.c [20-43]

```
20 void letesit(flex_tomb_t *a, index_t n) {
21     a->hatar = n;
22     if (n) {
23         int j;
24         a->lt = (elem_t**) malloc((1 + ((n - 1) / L)) * sizeof(lap_t));
25         for (j = 0; j <= ((n - 1) / L); ++j) {           /* lapok létesítése */
26             a->lt[j] = (elem_t*) malloc(L * sizeof(elem_t));
27         }
28     } else {
29         a->lt = NULL;
30     }
31 }
32
33 void megszuntet(flex_tomb_t *a) {
34     if (a->hatar) {
35         int j;
36         for (j = 0; j <= ((a->hatar - 1) / L); ++j) {     /* lapok törlése */
37             free(a->lt[j]);
38         }
39         free(a->lt);
40         a->lt = NULL;
41         a->hatar = 0;
42     }
43 }
```

Flexibilis tömb megvalósítás [4/4]

ftomb_t.c [45–66]

```
45 void novel(flex_tomb_t *a, index_t d) {
46     int j;
47     a->lt = (elem_t**) realloc(a->lt,
48                               (1 + ((a->hatar + d - 1) / L)) * sizeof(lap_t));
49     for (j = ((a->hatar) ? ((a->hatar - 1) / L) + 1 : 0);           /* új lapok */
50          j <= (a->hatar + d - 1) / L; ++j) {                     /* létesítése */
51         a->lt[j] = (elem_t*) malloc(L * sizeof(elem_t));
52     }
53     a->hatar += d;
54 }
55
56 void csokkent(flex_tomb_t *a, index_t d) {
57     if (d <= a->hatar) {
58         int j;
59         for (j = (a->hatar - d - 1) / L + 1; j <= (a->hatar - 1) / L; ++j) {
60             free(a->lt[j]);                                       /* felesleges lapok törlése */
61         }
62         a->hatar -= d;
63         a->lt = (elem_t**) realloc(a->lt,
64                                   (1 + ((a->hatar - 1) / L)) * sizeof(lap_t));
65     }
66 }
```

Rendezés több szempont szerint

Problémafelvetés és specifikáció

- Problémafelvetés

- A beolvasott adatokat rendezzük több szempont szerint is egy egyszerű rendezési algoritmussal és minden rendezés után legyen kiíratás is!

- Specifikáció

- Flexibilis tömbbel dolgozzunk.
- Az input név-adat párok sorozata, amelynek végét a * név jelzi.
- Az output a név szerint növekvő, adat szerint növekvő illetve csökkenő sorrendben rendezett három tömb.



Rendezés több szempont szerint

Algoritmustervezés – Szerkezeti ábra

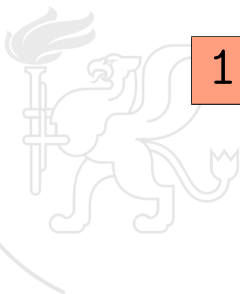
- A fő algoritmusban csak az elemeket kell beolvasni egy végjelig, majd rendre aktivizálni kell a különböző szempontok szerinti rendezést és ki kell íratni az eredményt.
- A rendezés a beszűrő rendezés lesz.



- Problémafelvetés
 - Rendezzük egy tömb elemeit!
- Specifikáció
 - Az input egy tömb, és a tömb elemtípusán értelmezett rendezési reláció.
 - Az output a megadott reláció alapján rendezett tömb.



- 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.

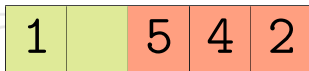


- A tömböt logikailag egy már rendezett és egy még rendezetlen részre osztjuk, és a rendezetlen rész első elemét beszúrjuk a rendezett elemek közé úgy, hogy azok rendezettek maradjanak.

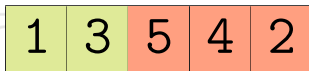


- A tömböt logikailag egy már rendezett és egy még rendezetlen részre osztjuk, és a rendezetlen rész első elemét beszurjuk a rendezett elemek közé úgy, hogy azok rendezettek maradjanak.

3



- 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.



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

5



- 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.



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

4

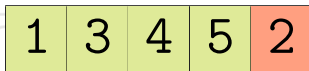


- A tömböt logikailag egy már rendezett és egy még rendezetlen részre osztjuk, és a rendezetlen rész első elemét beszúrjuk a rendezett elemek közé úgy, hogy azok rendezettek maradjanak.

4



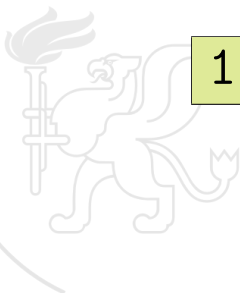
- A tömböt logikailag egy már rendezett és egy még rendezetlen részre osztjuk, és a rendezetlen rész első elemét beszúrjuk a rendezett elemek közé úgy, hogy azok rendezettek maradjanak.



- 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.

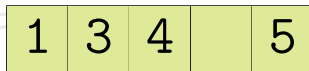
2

1 3 4 5



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

2



- 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.

2



1	3		4	5
---	---	--	---	---

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

2

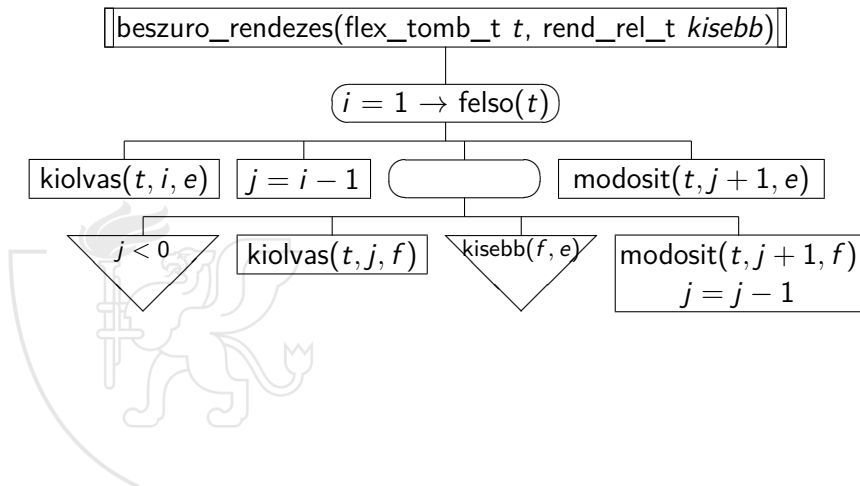
1 3 4 5

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



1	2	3	4	5
---	---	---	---	---

- A beszáró rendezés szerkezeti ábrája:



Rendezés több szempont szerint [1/8]

beszuro.c [1-25]

```
1 /* Rendezzük névsorba illetve átlag szerint a hallgatókat!
2  * Flexibilis tömbbel történik a megvalósítás, tehát a
3  * névsor hosszát nem kell előre megmondani.
4  * 1998. Február 16. Dévényi Károly, devenyi@inf.u-szeged.hu
5  * 2006. Augusztus 15. Gergely Tamás, gertom@inf.u-szeged.hu
6  * 2014. Október 15. Gergely Tamás, gertom@inf.u-szeged.hu
7  */
8
9 #include <stdio.h>
10 #include <string.h>
11 #include <stdlib.h>
12 #include <stdbool.h>
13
14 #define L 10                                     /* lapméret */
15
16 typedef struct {                                  /* a tömb elemtípusa */
17     char nev[21];
18     float adat;
19 } elem_t;
20
21 typedef elem_t *lap_t;
22
23 typedef lap_t *lapterkep_t;
24
25 typedef unsigned int index_t;
```

Rendezés több szempont szerint [2/8]

beszuro.c [27–50]

```
27 typedef struct flex_tomb_t {
28     lapterkep_t lt;
29     index_t      hatar;
30 } flex_tomb_t;
31
32 typedef bool (*rend_rel_t)(elem_t, elem_t);
33
34 /* A műveletek megvalósítása: */
35
36 void kiolvas(flex_tomb_t a, index_t i, elem_t *x) {
37     if (i < a.hatar) {
38         *x = a.lt[i / L][i % L];
39     }
40 }
41
42 void modosit(flex_tomb_t a, index_t i, elem_t x) {
43     if (i < a.hatar) {
44         a.lt[i / L][i % L] = x;
45     }
46 }
47
48 index_t felso(flex_tomb_t a) {
49     return a.hatar;
50 }
```

/ laptérkép */*
/ aktuális indexhatár */*

Rendezés több szempont szerint [3/8]

beszuro.c [52–75]

```
52 void letesit(flex_tomb_t *a, unsigned int n) {
53     a->hatar = n;
54     if (n) {
55         int j;
56         a->lt = (elem_t**) malloc(((1 + ((n - 1) / L)) * sizeof(lap_t)));
57         for (j = 0; j <= ((n - 1) / L); ++j) {           /* lapok létesítése */
58             a->lt[j] = (elem_t*) malloc(L * sizeof(elem_t));
59         }
60     } else {
61         a->lt = NULL;
62     }
63 }
64
65 void megszuntet(flex_tomb_t *a) {
66     if (a->hatar) {
67         int j;
68         for (j = 0; j <= ((a->hatar - 1) / L); ++j) {     /* lapok törlése */
69             free(a->lt[j]);
70         }
71         free(a->lt);
72         a->lt = NULL;
73         a->hatar = 0;
74     }
75 }
```

Rendezés több szempont szerint [4/8]

beszuro.c [77–98]

```
77 void novel(flex_tomb_t *a, index_t d) {
78     int j;
79     a->lt = (elem_t**) realloc(a->lt,
80                               (1 + ((a->hatar + d - 1) / L)) * sizeof(lap_t));
81     for (j = ((a->hatar) ? ((a->hatar - 1) / L) + 1 : 0); /* új lapok */
82          j <= (a->hatar + d - 1) / L; ++j) { /* létesítése */
83         a->lt[j] = (elem_t*) malloc(L * sizeof(elem_t));
84     }
85     a->hatar += d;
86 }
87
88 void csokkent(flex_tomb_t *a, index_t d) {
89     if (d <= a->hatar) {
90         int j;
91         for (j = (a->hatar - d - 1) / L + 1; j <= (a->hatar - 1) / L; ++j) {
92             free(a->lt[j]); /* felesleges lapok törlése */
93         }
94         a->hatar -= d;
95         a->lt = (elem_t**) realloc(a->lt,
96                                   (1 + ((a->hatar - 1) / L)) * sizeof(lap_t));
97     }
98 }
```

Rendezés több szempont szerint [5/8]

beszuro.c [100–122]

```
100 /* Rendezés */
101
102 void beszuro_rendezes(flex_tomb_t t, rend_rel_t kisebb) {
103     /* A kisebb rendezési reláció szerinti helyben rendezés */
104     int i, j;
105     elem_t e, f;
106     for (i = 1; i < felso(t); ++i) {
107         kiolvas(t, i, &e);
108         j = i - 1;
109
110         while (true) {
111             if (j < 0) {
112                 break;
113             }
114             kiolvas(t, j, &f);
115             if (kisebb(f, e)) {
116                 break;
117             }
118             modosit(t, ((j--) + 1), f);
119         }
120         modosit(t, j + 1, e);
121     }
122 } /* beszuro_rendezes */
```

Rendezés több szempont szerint [6/8]

beszuro.c [124–147]

```
124 bool rend_nev_novekvo(elem_t x, elem_t y) {
125                                     /* a névsor szerinti rendezési reláció */
126     return strcmp(x.nev, y.nev) <= 0;
127 }
128
129 bool rend_adat_novekvo(elem_t x, elem_t y) {
130                                     /* az adat szerinti rendezési reláció */
131     return x.adat <= y.adat;
132 }
133
134 bool rend_adat_csokkeno(elem_t x, elem_t y) {
135                                     /* az adat szerint csökkenő rendezési reláció */
136     return x.adat >= y.adat;
137 }
138
139 void kiiras(flex_tomb_t t) {
140                                     /* Kiírás */
141     elem_t e;
142     index_t i;
143     for (i = 0; i < felso(t); ++i) {
144         kiolvas(t, i, &e);
145         printf("%6.2f□%s\n", e.adat, e.nev);
146     }
147 }
```

Rendezés több szempont szerint [7/8]

beszuro.c [149–165]

```
149 int main() {
150     flex_tomb_t sor;
151     elem_t      hallg;           /* beolvasáshoz */
152     index_t     i;
153
154     letesit(&sor, 0);           /* a flexibilis tömb létesítése */
155                                 /* beolvasás */
156     printf("Kérem az adatsort, külön sorban név és adat!\n");
157     printf("A végét a * jelzi.\n");
158     scanf("%20[^\n]*[^\n]", hallg.nev); getchar();
159     i = 0;                      /* az i. helyre fogunk beírni */
160     while (strcmp(hallg.nev, "*") {
161         novel(&sor, 1);         /* a flexibilis tömb bővítése */
162         scanf("%f*[^\n]", &hallg.adat); getchar();
163         modosit(sor, i++, hallg);
164         scanf("%20[^\n]*[^\n]", hallg.nev); getchar();
165     }
```



Rendezés több szempont szerint [8/8]

beszuro.c [167–181]

```
167     beszuro_rendezes(sor, rend_nev_novekvo);           /* Rend. névsor szerint */
168     printf("Névsor_szerint_rendezeve:\n");
169     kiiras(sor);                                       /* Kiíratás */
170
171     beszuro_rendezes(sor, rend_adat_novekvo);         /* Rend. adat szerint */
172     printf("Adat_szerint_rendezeve:\n");
173     kiiras(sor);                                       /* Kiíratás */
174
175     beszuro_rendezes(sor, rend_adat_csokkeno);        /* Rendezés újra */
176     printf("Adat_szerint_csökkenő_sorba_rendezeve:\n");
177     kiiras(sor);                                       /* Kiíratás */
178
179     megszuntet(& sor);                                /* a flexibilis tömb törlése */
180     return 0;
181 }
```



- 1 **Bemutakozás**
- Kurzus információk
 - A SZTE és az informatikai képzés

- 2 **Linux**
- Alapfogalmak
 - Linux parancsok
 - Linux shell
 - Felhasználók
 - Hálózat

- 3 **Gyors C áttekintés**
- Bevezető
 - Pénzváltás (1. verzió)
 - Pénzváltás (2. verzió)
 - Röppálya számítás
 - Röppálya szimuláció
 - Az év napja
 - Csúszóátlag adott elemszámmra
 - Csúszóátlag parancssorból
 - Basename standard inputról
 - Basename parancssorból
 - Tér legtávolabbi pontjai
 - A nappalis gyakorlat értékelése

- 4 **Alapok**
- Alapfogalmak
 - A programozás fázisai
 - Algoritmus vezérlése
 - A C nyelvű program
 - Szintaxis
 - A C nyelv elemi adattípusai
 - A C nyelv utasításai

- 5 **Vezérlési szerkezetek**
- Bevezetés
 - Szekvenciális vezérlés
 - Függvények
 - Szelekciós vezérlések
 - Ismétléses vezérlések 1.
 - Eljárásvezérlés
 - Ismétléses vezérlések 2.

- 6 **Folyamatábra és struktúradiagram**

- 7 **Adatszerkezetek**
- Az adatkezelés szintjei
 - Elemi adattípusok
 - Pointer adattípus
 - Tömb adattípus

- Sztringek
- Pointerek és tömbök C-ben
- Rekord adattípus
- Függvény pointer
- Halmaz adattípus
- Flexibilis tömbök
- **Láncolt listák**
- Típusokról C-ben

- 8 **IO**
- Alapok
 - Adatállományok

- 9 **C fordítás**
- A fordítás folyamata
 - A preprocesszor
 - A C fordító
 - Assembler
 - Linker és modulok

- 10 **Gyakorlati kérdések**
- Memóriahasználat
 - Gyakori C hibák
 - `where.c` felboncolva

- A tömb egy hatékony adattípus elemek sorozatának tárolására.
- De mi a helyzet, ha nekünk elemek olyan sorozata kell, amelyen a beszúrás és törlés műveletét (is) szeretnénk értelmezni?
- Akkor a *Lista* adatszerkezet kell használnunk.
- Jelöljük az *elemtip* alaptípusból képzett *Lista* típust *lista*-val, és a lista egy helyét meghatározó típust *pozicio*-val.



Lista absztrakt adattípus műveletei

- **Létesít**($\leftarrow l:lista$)
 - Egy üres lista létesítése és visszaadása az l változóban.
- **Megszüntet**($\leftrightarrow l:lista$)
 - Az l lista megszüntetése.
- **Kiolvas**($\rightarrow p:pozicio; \leftarrow x:elemtip$)
 - A p pozíción lévő érték kiolvasása adott x , *elemtip* típusú változóba.
- **Módosít**($\rightarrow p:pozicio; \rightarrow x:elemtip$)
 - A p pozíción lévő eleme értékének módosítása x -re.
- **Beszúr**($\leftrightarrow l:lista; \rightarrow p:pozicio; \rightarrow x:elemtip$)
 - Az adott l listába a p pozíció elé szúrjuk be az x értéket.
- **Töröl**($\leftrightarrow l:lista; \rightarrow p:pozicio$)
 - Töröljük az l lista p pozícióján lévő elemét.
- **Következő**($\rightarrow l:lista; \rightarrow p:pozicio; \leftarrow n:pozicio$)
 - Az l lista p pozícióját követő pozíciót adja vissza n -ben.
- **Értékadás**($\leftarrow lhs:lista; \rightarrow rhs:lista$)
 - Értékadó művelet: az lhs változó felveszi az rhs kifejezés értékét.

Listák megvalósítása

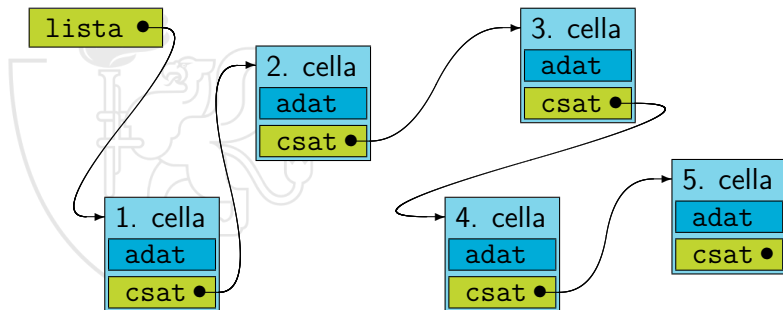
Láncolt listák

- A tömb egy nagyon hatékony adattípus, ami az egyes elemek elérését és módosítását illeti. A flexibilis tömbök egy kisebb hatékonyságcsökkenés árán a dinamikus méretmódosításokat is lehetővé teszik.
- De mi a helyzet, ha a lista beszúrás és törlés műveletét szeretnénk olcsón elvégezni? A tömb ilyenkor nyilván nem hatékony.
- A *Láncolt lista* adatszerkezet viszont éppen a beszúrás-törlés műveletben hatékony, és kevésbé jó választás, ha az elemeknek nem csak a sorrendje érdekes de a sorszámuk alapján is be szeretnénk őket azonosítani.

Láncolt listák

Virtuális adattípus

- A láncolt listában az adatokat olyan cellákban tároljuk, ahol a cella tartalmazza a következő cella elérési információit.
- Az elérési információ általában a sorozat következő elemét tartalmazó cellára (dinamikus változóra) mutató pointer.
- Magát a sorozatot az első elemére mutató pointerrel adhatjuk meg, hiszen az első elem ismeretében a lista összes eleme elérhetővé válik.



Láncolt lista megvalósítás [1/4]

lanc_t.h [1–22]

```
1 #ifndef LANC_H
2 #define LANC_H
3
4 typedef ???      elem_t;          /* a lista_t elemtípusa */
5 typedef struct cella_t {
6     elem_t adat;                  /* adatalem */
7     struct cella_t *csat;         /* a következő elem */
8 } cella_t;
9 typedef struct cella_t *pozicio_t;
10 typedef pozicio_t lista_t;
11
12 /* A műveletek deklarációja: */
13 void letesit(lista_t *l);
14 void megszuntet(lista_t *l);
15 void kiolvas(pozicio_t p, elem_t *x);
16 void modosit(pozicio_t p, elem_t x);
17 void beszur(lista_t *l, pozicio_t p, elem_t x);
18 void torol(lista_t *l, pozicio_t p);
19 void kovetkezo(pozicio_t p, pozicio_t *n);
20 void ertekadas(lista_t *lhs, lista_t rhs);
21
22 #endif // LANC_H
```

Láncolt lista megvalósítás [2/4]

lanc_t.c [1–23]

```
1#include <stdlib.h>
2#include "lanc_t.h"
3
4void letesit(lista_t *l) {
5    *l = NULL;
6}
7
8void megszuntet(lista_t *l) {
9    pozicio_t p = *l;
10   while(p) {
11       p = p->csat;
12       free(*l);
13       *l = p;
14   }
15}
16
17void kiolvas(pozicio_t p, elem_t *x) {
18    *x = p->adat;
19}
20
21void modosit(pozicio_t p, elem_t x) {
22    p->adat = x;
23}
```


Láncolt lista megvalósítás [3/4]

lanc_t.c [25–51]

```
25 void beszur(lista_t *l, pozicio_t p, elem_t x) {
26     pozicio_t q = *l;
27     if(q == p) {
28         q = *l = malloc(sizeof(cella_t));
29     } else {
30         while(q->csat != p) {
31             q = q->csat;
32         }
33         q->csat = malloc(sizeof(cella_t));
34         q = q->csat;
35     }
36     q->csat = p;
37     q->adat = x;
38 }
39
40 void torol(lista_t *l, pozicio_t p) {
41     pozicio_t q = *l;
42     if(q == p) {
43         *l = p->csat;
44     } else {
45         while(q->csat != p) {
46             q = q->csat;
47         }
48         q->csat = p->csat;
49     }
50     free(p);
51 }
```

Láncolt lista megvalósítás [4/4]

lanc_t.c [53–71]

```
53 void kovetkezo(pozicio_t p, pozicio_t *n) {
54     *n = p->csat;
55 }
56
57 void ertekadas(lista_t *lhs, lista_t rhs) {
58     megszuntes(lhs);
59     if(!rhs) {
60         *lhs = NULL;
61         return;
62     }
63     pozicio_t p;
64     *lhs = p = malloc(sizeof(cella_t));
65     p->adat = rhs->adat;
66     while((rhs = rhs->csat)) {
67         p = p->csat = malloc(sizeof(cella_t));
68         p->adat = rhs->adat;
69     }
70     p->csat = NULL;
71 }
```



- A fenti definíciók esetén egy adott p pozíción az eltárolt adatra a $p \rightarrow \text{adat}$, a következő pozícióra pedig a $p \rightarrow \text{csat}$ kifejezésekkel hivatkozhatunk.
- Megjegyzendő, hogy a $.$ és \rightarrow mezőkiválasztás operátorok a precedencia-hierarchia csúcsán állnak, és ezért nagyon szorosan kötnek.
 - A $++p \rightarrow \text{adat}$ például nem a p pointert, hanem az adat mezőt inkrementálja, mivel az alapértelmezés szerinti zárójelezés a $++(p \rightarrow \text{adat})$.
 - A $(++p) \rightarrow \text{adat}$ már a p inkrementálása után férne hozzá az adat mezőhöz, de ahhoz, hogy ez helyesen működjön, egy sokkal bonyolultabb megvalósítás lenne szükséges, mert az inkrementáló művelet nincs tekintettel arra, hogy mi éppen egy lánc egyik pozíciójára alkalmaztuk.

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

```
1 typedef void (*muvelettip)(elemtip*);
2 void bejar(lista l, muvelettip muv)
3 {
4     for(; l != NULL; l = l->csat) {
5         /* művelet a sorozat elemén */
6         muv(&(l->adat));
7     }
8 }
```

- Ezzel a módszerrel a megvalósítás egyes technikai részleteit jobban elkülöníthetjük, érthetőbbé, átláthatóbbá, könnyen módosíthatóbbá válik a kódunk.
 - Ha például adatszerkezetet cserélünk, a bejárását elegendő a bejar() függvényben egyetlen helyen átírni, hogy a program továbbra is jól működjön.

- 1 **Bemutakozás**
- Kurzus információk
 - A SZTE és az informatikai képzés

- 2 **Linux**
- Alapfogalmak
 - Linux parancsok
 - Linux shell
 - Felhasználók
 - Hálózat

- 3 **Gyors C áttekintés**
- Bevezető
 - Pénzváltás (1. verzió)
 - Pénzváltás (2. verzió)
 - Röppálya számítás
 - Röppálya szimuláció
 - Az év napja
 - Csúszóátlag adott elemszámmra
 - Csúszóátlag parancssorból
 - Basename standard inputról
 - Basename parancssorból
 - Tér legtávolabbi pontjai
 - A nappalis gyakorlat értékelése

- 4 **Alapok**
- Alapfogalmak
 - A programozás fázisai
 - Algoritmus vezérlése
 - A C nyelvű program
 - Szintaxis
 - A C nyelv elemi adattípusai
 - A C nyelv utasításai

- 5 **Vezérlési szerkezetek**
- Bevezetés
 - Szekvenciális vezérlés
 - Függvények
 - Szelekciós vezérlések
 - Ismétléses vezérlések 1.
 - Eljárásvezérlés
 - Ismétléses vezérlések 2.

- 6 **Folyamatábra és struktúradiagram**
- 7 **Adatszerkezetek**
- Az adatkezelés szintjei
 - Elemi adattípusok
 - Pointer adattípus
 - Tömb adattípus

- Sztringek
- Pointerek és tömbök C-ben
- Rekord adattípus
- Függvény pointer
- Halmaz adattípus
- Flexibilis tömbök
- Láncolt listák
- **Típusokról C-ben**

- 8 **IO**
- Alapok
 - Adatállományok

- 9 **C fordítás**
- A fordítás folyamata
 - A preprocesszor
 - A C fordító
 - Assembler
 - Linker és modulok

- 10 **Gyakorlati kérdések**
- Memóriahasználat
 - Gyakori C hibák
 - where.c felboncolva

Bonyolultabb deklarációk

- A C nyelvben az összetett típusok megadásakor megismert műveleteket (pl. pointer, tömb vagy függvény képzés) véges sokszor alkalmazhatjuk.
- Ezen műveletek között a típusképzés során ugyanolyan precedencia áll fent mint a nekik megfelelő dereferencia (*), tömbelem-kiválasztás ([]) és függvényhívás (()) operátorok között, és ezt még az asszociativitás iránya is bonyolítja. Ráadásul ezek zárójelezéssel ugyanúgy felülírhatóak, mint a kifejezésekben.
- Érdemes tehát néhány példával megvilágítani a bonyolultabb eseteket:

```
int *p      /* int-re mutató pointer */
int x[10]   /* tömb 10 int-ből */
int (*x)[10] /* pointer mutat egy tömbre, ami
              10 int-ből áll */
int *x[10]  /* tömb 10 pointerből, melyek
              int-re mutatnak */
```

- Nézzünk példát függvényekkel is:

```
void (*f)(int)    /* pointer int paraméterű eljárásra */  
int (*f)(void)    /* pointer egy paraméter nélküli int  
                    visszatérési értékű függvényre */  
int (*x[])(int)  /* int visszatérési értékű és int  
                    paraméterű függvényre mutató  
                    pointerek tömbje */  
char ((*f())[2])() /* char típusú függvényre mutató  
                    pointereket tartalmazó kételemű  
                    tömbre mutató pointerrel  
                    visszatérő függvény */  
char ((*x[3])())[5] /* öt elemű char tömbre mutató  
                    pointerrel visszatérő függvé-  
                    nyekre mutató pointerekből álló  
                    3 elemű tömb */
```

Bonyolultabb deklarációk

- Ilyen deklarációkat meg lehet próbálni egyszerre felírni, de ehhez sok C programozási tapasztalat kell.
- A cél típust el lehet érni típusdefiníciók sorozatán keresztül is:

```
typedef char a[5];  
typedef a *pa;  
typedef pa (*fp_pa)();  
typedef fp_pa x[3];
```

- Illetve használjuk a cdecl programot ha angolul értünk vagy el tudjuk mondani mit szeretnénk.

```
cdecl> explain char ((*x[3])())[5]  
declare x as array 3 of pointer to function returning pointer to array 5 of char  
cdecl> declare x as array 3 of pointer to function returning pointer to array 5 of char  
char ((*x[3])())[5]
```


Bonyolultabb deklarációk

- Egy deklarációt talán belülről kifelé haladva könnyebb megérteni.
- Van viszont olyan eset, amikor magát a típust kell megadnunk, így a belülről kifelé megértéshez nincs meg az azonosító, mint kiindulópont.
- Ilyen például a típuskényszerítés, vagy amikor a `sizeof` argumentumaként típust adunk meg.
- A típus megadása ilyenkor szintaktikailag ugyanúgy történik mint a változó deklarálása, csak az azonosító marad el.

```
int *  
int [10]  
int (*)[10]  
int *[10]  
void (*)(int)  
int (*)(void)  
int (*[])(int)  
char ((*()) [2])()  
char ((*[3])()) [5]
```

Típuskényszerítés

- Típuskényszerítésről akkor beszélünk, amikor egy adott típusú értéket más típusú értéként szeretnénk kezelni.
- Például két egész érték valós hányadosára vagyunk kíváncsiak. Ilyen esetben még az osztás művelet előtt valamelyik értéket valós típusúvá kell tenni:

```
int i = 7;
int j = 2;
double x;
x = i / j;           /* x értéke 3.0 lesz */
x = (double) i / j; /* x értéke 3.5 lesz */
```

- Gyakran használjuk még a malloc() függvénnyel kapcsolatban:

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

Típuskényszerítés

- A típuskényszerítés egy konverziós művelet, tehát nem alkalmazható mindig. Egy `void *p` pointerre például nem alkalmazható így:

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

De így már igen:

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

- Vigyázzunk mert nem minden típuskényszerítés ad értelmes eredményt:

```
double d = 1.0;  
unsigned long long int *p;  
p = (unsigned long long int *)&d;  
/* *p == 0x3ff0000000000000 */
```

C műveletek prioritása

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