Imperatív programozás

Típusok

Kozsik Tamás és mások

1 Konstansok

Konstansok C-ben

Konstansok

```
const int i = 3;
int const j = 3;

const int t[] = {1,2,3};


const int *p = &i;

int v = 3;
int * const q = &v;
```

Fordítási hiba

```
i = 4;
j = 4;

t[2] = 4;
t = {1,2,4};

*p = 4;


q = (int *)malloc(sizeof(int));
```

Mit is jelent ez a deklaráció?

```
int const * a, b;
```

Nem teljes a biztonság

```
const int i = 3;
const int *r = &i;      /* korrekt */

. . .

int j = 2;
r = &j;                 /* korrekt */

. . .

r = &i;
int * p = r;            /* csak warning */
```

```
int * const q = &i;     /* csak warning */

*p = *q = 4;            /* i megváltozik :-( */
```

Karakter első előfordulása szövegben

```
char *strfind( char *str, int c ){
    int i = 0;
    while( str[i] != 0 && str[i] != c ) ++i;
    return &str[i];
}


char p[] = "Hello";
char *q = strfind(p,'e');
*q = 'o';                    /* ok: p-ben "Hollo", q-ban "ollo" */
...
char *r = strfind("Hello",'e');   /* módosíthatatlan tárhelyen! */
*r = 'o';                         /* Segmentation fault :-( */
```

Szövegliterálok módosíthatatlan tárhelyen

```
char *r = "Hello";
*r = 'h';                   /* segmentation fault :-( */
...
const char *cr = "Hello";
*cr = 'h';                  /* fordítási hiba :-) */
```

Zárjuk ki az illegális memóriamódosítást!

```
const char *strfind( char *str, int c ){
    int i = 0;
    while( str[i] != 0 && str[i] != c ) ++i;
    return &str[i];
}

const char *r = strfind("Hello",'e');
*r = 'o';                   /* fordítási hiba :-) */
...
char p[] = "Hello";
char *q = strfind(p,'e');   /* fordítási hiba :-( */
*q = 'o';
...
const char *r = "Hello";
*r = strfind(r,'e');        /* fordítási hiba :-( */
```

Működjön const-ra?

```
const char *strfind( const char *str, int c ){
    int i = 0;
    while( str[i] != 0 && str[i] != c ) ++i;
```

```
    return &str[i];
}

const char *r = "Hello";
r = strfind(r,'e');
*r = 'o';                   /* fordítási hiba :-) */

...

char p[] = "Hello";
char *q = strfind(p,'e');
*q = 'o';                   /* fordítási hiba :-( */
```

const-ra polimorf megoldás nincs

```
char *strfind( char *str, int c ){
    int i = 0;
    while( str[i] != 0 && str[i] != c ) ++i;
    return &str[i];
}

const char *conststrfind( const char *str, int c ){
    int i = 0;
    while( str[i] != 0 && str[i] != c ) ++i;
    return &str[i];
}
```

- Kód duplikációja?
- Karakterliterállal az első még mindig meghívható

Ugyanez a string szabványos könyvtárban

```
char *strchr( const char *str, int c ){
    while( *str != 0 && *str != c ) ++str;
    return str;
}

...
```

- Általánosan meghívható
- Karakterliterállal meghívható

Ugyanez a string szabványos könyvtárban

```
char *strchr( const char *str, int c ){ ... }

const char *p = "Hello";
char *r = "Hello";
char q[] = "Hello";

p = strchr(p,'e');
r = strchr(q,'e');

*p = 'o';           /* fordítási hiba :-) */
*r = 'o';           /* ok :-) */

r = strchr(p,'o');  /* vagy strchr("Hello",'e') */
*r = 'e';           /* segmentation fault :-( */
```

2 Típusszinonímák

typedef a C-ben

```
typedef double Element;

...

Element max( Element array[], int size ){
    Element m = array[0];
    while( size > 0 ){
        --size;
        if( array[size] > m )
            m = array[size];
    }
    return m;
}

...
```

- Mint a Haskellben a `type`
- Típusszinoníma: ugyanannak a típusnak több neve is van
- Esztétikai szerepe van
- Karbantarthatóság (olvashatóság, módosíthatóság)

3 Típuskonstrukciók

Típusok, típuskonstrukciók

- Egyszerű típusok
  - Szám típusok
    - Egész típusok (és karakter típusok)
    - Lebegőpontos típusok
  - Felsorolási típusok
  - Mutató típusok
- Összetett típusok
  - Tömb
  - Lista (Python, Haskell)
  - Rendezett n-es (Python, Haskell)
  - Halmaz (Python `set` és `frozenset`)
  - Leképezés (Python `dictionary`)
  - Rekord (C `struct`, Haskell record)
  - Unió típus (C `union`, Haskell algebrai adattípus)
  - Osztály

Sorozat típusnak nevezzük azt a típust, amelynek típusértékei több, (jellemzően) azonos típusba tartozó értékből állnak. Az egyes elemekre indexeléssel szoktunk hivatkozni. Ilyen sorozat típusnak tekintjük a C tömbjeit, valamint a Python listáit és akár a tuple-jeit is.

Mivel a Python az értékekhez sok esetben referenciák segítségével biztosít hozzáférést, és dinamikusan típusos, megengedi azt, hogy a listákban és tuple-ökben különféle típusú értékek szerepeljenek elemként. Valójában a listák és a tuple-ök *referenciák sorozatai*.

Egy lényeges különbség (már tanultuk) a Python lista és tuple között az, hogy az előbbi módosítható (mutable), az utóbbi pedig módosíthatatlan (immutable) adatszerkezet.

Korábban már tanultunk a Python gyakran használt adatszerkezeteiről. A sorozatok (listák és tuple-ök) mellett kiemelendők a halmazok (`set` és `frozenset` – az előbbi módosítható, az utóbbi módosíthatatlan), valamint a leképezések (dictionary, módosítható).

3.1 Felsorolási típusok

Felsorolási típus

Haskell: algebrai adattípussal

```
data Color = White | Green | Yellow | Red | Black
```

C: enum

```
enum color { WHITE, GREEN, YELLOW, RED, BLACK };

...

const char* property( enum color code ){
    switch( code ){
        case WHITE:  return "pure";
        case GREEN:  return "jealous";
        case YELLOW: return "envy";
        case RED:    return "angry";
        case BLACK:  return "sad";
        default:     return "?";
    }
}
```

A felsorolási típusok olyan típusok, amelyeknek a típusértékeit egy felsorolással adjuk meg. Ilyen típusokat sok nyelvben lehet definiálni. Találkozhattunk vele például a Haskellben, ahol algebrai adattípusokkal (`data` konstrukcióval) fejezhetjük ki a felsorolási típusokat. (A típusértékek ebben az esetben a típus „adatkonstruktorai" lesznek.)

A C nyelvben is definiálhatunk felsorolási típusokat az `enum` kulcsszóval. A fenti példában a `color` egy felsorolási típus lesz. Egy fontos tulajdonsága a felsorolási típusoknak, hogy használhatjuk őket `switch` utasításhoz. Ez azért van, mert a felsorolási típusok belül egy egész típussal vannak reprezentálva. Ezért aztán minden műveletet, amit egész számokkal végezhetünk, végezhetjük felsorolási típusokkal is. Más nyelvek (pl. Haskell) szigorúbbak szoktak lenni, és nem keverik össze az egész számok típusát a felsorolási típusokkal, és persze nem engedik egymással összekeveredni a különböző felsorolási típusok értékeit sem. A C nyelv túl lazán bánik ezzel a konstrukcióval!

Vegyük még azt is észre, hogy amikor használjuk a felsorolási típust, az `enum` kulcsszót akkor is kiírjuk! A `typedef`-fel ezen segíthetünk, és az `enum color` típust közelebb hozhatjuk a Haskelles megfelelőjéhez.

enum és typedef

```
enum color { WHITE, GREEN, YELLOW, RED, BLACK };

typedef enum color Color;

const char* property( Color code ){
    switch( code ){
        case WHITE:  return "pure";
        case GREEN:  return "jealous";
        case YELLOW: return "envy";
        case RED:    return "angry";
        case BLACK:  return "sad";
        default:     return "?";
    }
}
```

Ebben a példában a `typedef` kulcsszó segítségével egy könnyebben használható típusnevet vezetünk be az `enum color` alternatívájaként: `Color`. A `typedef` kicsit félrevezető elnevezés. Nem egy új típust definiálunk vele, hanem egy meglévő típushoz (pl. `enum color`) egy új nevet, szinonímát vezethetünk be

a segítségével: azaz *deklarálhatunk*, nevet rendelhetünk egy entitáshoz ezzel a kulcsszóval. (A `typedef` ennélfogva a Haskell `type` konstrukciójának felel meg.)

enum: valójában egy egész szám típusra képződik le

```
enum color { WHITE, GREEN, YELLOW, RED, BLACK };

const char * const properties[] =
    { "pure","jealous","envy","angry","sad" };

const char* property( enum color code ){
    return properties[code];
}
```

Trükközés a típusértékekkel

```
enum color { WHITE = 1, GREEN, YELLOW, RED = 6, BLACK };

typedef enum color Color;
const char* property( Color code ){ ... }

...
int main( int argc, char *argv[] )
{
    for( --argc; argc>0; --argc )
    {
       printf("%s\n", property( atoi(argv[argc]) ));
    }
    return 0;
}
```

Ezen a példán azt is megfigyelhetjük, hogy a felsorolási típus értékeit reprezentáló egész számot meg is adhatjuk a típus definíciójában. Ha nem adunk meg értéket, nullától szoktak számozódni a típusértékek, és egyesével növekednek. Ha valamelyik típusértékhez hozzárendelünk egy egész számot, a rá következő típusértékhez eggyel nagyobb egész szám rendelődik. A fenti példában a `YELLOW` ezért 3, a `BLACK` pedig `7` lesz.

A `main` függvényben a parancssori argumentumokat (amelyek sztringek) az `stdlib` könyvtárban definiált `atoi` függvénnyel konvertáljuk át egész számmá, és az így kapott egész számokat feleltetjük meg aktuális paraméterként a `property` függvény `enum color` típusú formális paraméterének. Ez is az egész és felsorolási típusok közötti könnyű átjárhatóságot (és egyben a felsorolási értékek viszonylag gyenge típusozottságát) illusztrálja.

3.2 Rekord típusok

Direktszorzat típusok

(Potenciálisan) különböző típusú elemekből konstruált összetett típus

- tuple
- rekord, struct

C struct

```
struct month { char *name; int days; };  /* típus létrehozása */

struct month jan = {"January", 31};      /* változó létrehozása */
```

```c
/* three-way comparison */
int compare_days_of_month( struct month left, struct month right )
{
    return left.days - right.days;
}
```

A sorozatok mellett az összetett típusok egy másik nagy családját alkotják azok az összetett típusok, amelyeknek az értékei több, potenciálisan különböző típusú elemekből állnak össze. A Haskell nyelvből megismert *rendezett n-es* (tuple) egyértelműen ide sorolható. Az egyes elemekhez a Haskell nyelvben mintaillesztéssel, valamint könyvtári függvényekkel férhetünk hozzá. Ugyanezt a technikát láthatjuk a Python tuple-jei esetén is.

```python
v = (1, "-szerű")
n, str = v
```

A Python az indexelés műveletét is biztosítja a tuple-ökhöz, így ezt az adatstruktúrát nyugodt szívvel sorolhatjuk a sorozatok és a direktszorzatok közé is.

Sok programozási nyelv a direktszorzat típusok létrehozásához a rekord konstrukciót ajánlja. A rekord annyiban más, mint a rendezett n-es, hogy az egyes elemekhez *szelektorokkal* férünk hozzá. Amikor a rekord típust definiáljuk, megadjuk, hogy milyen nevű és típusú *mezőkből* épül fel. Az egyes mezőkhöz (elemekhez) a mezőnév használható szelektorként.

A C nyelvben a `struct` kulcsszóval vezethetünk be rekordokat. A fenti `struct month` rekord két mezője sztring, illetve `int` típusú, a mezők neve `name`, illetve `days`.

A rekord típus leggyakoribb implementációja, hogy az egyes mezők egymás után helyezkednek el a memóriában. Minden mező a rekord elejéhez képest saját távolsággal (offset) rendelkezik. Esetenként azonban a mezők között lehetnek „lyukak” (gap, padding) is, itt nem tárolunk információt. Lyukak amiatt lehetnek, mert egyes fordítók bizonyos típusokat csak adott (például néggyel osztható) bájtcímekre helyeznek el (gépi szó határaira igazítják). Ebből következően a rekord mérete nagyobb vagy egyenlő a mezők méreteinek összegével.

```c
struct month { char short_name; char *name; int days; };
printf("%ld\n", sizeof(struct month));   /* nálam 24-et írt ki */
```

A rekord típussal rendszerint csak a legegyszerűbb műveleteket végezhetjük el, pl.

- az értékadást, ide értve az érték szerinti paraméterátadást és függvényvisszatérést is,
- a rekord címének lekérdezését, és
- az egyes mezők elérését.

Miután a rekord egy mezőjét elértük, az adott mező típusának megfelelő műveleteket végezhetjük el rajta.

C struct

```c
struct month { char *name; int days; };
struct month jan = {"January", 31};

struct date { int year; struct month *month; char day; };
struct person { char *name; struct date birthdate; };

typedef struct person Person;

int main(){
   Person pete = {"Pete", {1970,&jan,28}};
   printf("%d\n", pete.birthdate.month->days);
   return 0;
}
```

Nézzünk egy picit összetettebb példát. A korábban már látott `struct month` után hozzuk létre a `struct date` és a `struct person` struktúrákat is. Az utóbbi tartalmaz mezőként egy `struct date` típusú értéket, mutatva, hogy az összetett típusok tetszőleges mélységben egymásba is ágyazhatók.

A `struct date` struktúrát lehetett volna három `int` típusú mezővel is definiálni – hiszen a rekordoknál az is megengedett, hogy a mezők ugyanolyan típusúak legyenek. Igazából koncepcionális kérdés az, hogy egy három `int`-ből álló adatot tömb vagy struktúra segítségével írunk le. Az előbbi elemeit indexeléssel, az utóbbi elemeit a mezőnevekkel érhetjük el. Egy dátum típus esetében a struktúra jobb megoldásnak tűnik.

Azt figyelhetjük meg a `struct date` struktúrában, hogy a második mező egy mutató egy `struct month` struktúrára. Amikor a `main`-ben létrehozzuk a `pete` változót, akkor az inicializáció során a `jan` változóra (struktúrára) mutató pointert használjuk a születési idő hónapjaként. A gyakorlatban sokszor előfordul, hogy egy struktúrát mutatókon keresztül érünk el. A mutató dereferálása és a struktúra egy mezőjének kiválasztása nagyon kényelmes a `->` operátor használatával.

```
struct month *next_month = &jan;
char        *name = (*next_month).name;
char *alternative =   next_month->name;
```

A kétfajta hivatkozás ekvivalens. Vegyük észre, hogy a `(*next_month).name` kifejezésből a zárójel nem hagyjató el!

A `typedef` konstrukcióval ugyanúgy új nevet vezethetünk be a struktúra típusokhoz, mint a felsorolási típusokhoz. Ezzel a trükkel ismét elérhetjük, hogy a típusunkhoz a `struct` kulcsszó használata nélkül is hozzáférjünk.

Paraméterátadás

```
void one_day_forward( struct date *d ){
    if( d->day < d->month->days ) ++(d->day);
    else { ... }
}

struct date next_day( struct date d ){
    one_day_forward(&d);
    return d;
}

int main(){
   struct date new_year = {2019, &jan, 1};
   struct date sober;
   sober = next_day(new_year);
   return ( sober.day != 2 );
}
```

Struktúrákat érték szerint adhatunk át paraméterként, és érték szerint kapjuk vissza őket visszatérési értékként. Ez nagyon más, mint a tömbök esetén. Egy tömb átadása a kezdőcímének (egy mutató) átadását jelenti: erre mondtuk azt, hogy *megosztás-szerinti paraméterátadás*. Tömböket visszaadni pedig nem is lehet a C-ben. (Mutatókat természetesen visszaadhatunk.)

A fenti `one_day_forward` függvény egy struktúrára mutató pointert vár paraméterként, ezzel szimuláljuk a cím-szerinti paraméterátadást. A függvény meg szeretné változtatni azt a dátum objektumot, amelynek címét paraméterként kapta. Ha nem mutatót adtunk volna át paraméterként, akkor a függvény az érték-szerinti paraméterátadás miatt csak egy lokális változót módosított volna, nem a hívás helyén elérhető dátum objektumot. Nyilván nem ez a cél ebben az esetben, hiszen itt egy be- és kimenő szemantikájú paraméterátadást akarunk megvalósítani: szeretnénk, ha a `one_day_forward` függvénnyel egy dátumot meg tudnánk változtatni. (Struktúrákat akkor is szoktunk mutatón keresztül átadni paraméterként, ha a hívott függvény nem akarja megváltoztatni a struktúrát. Akkor szoktunk ilyet csinálni, ha nagy méretű a struktúra, és el szeretnénk kerülni az érték-szerinti paraméterátadással járó másolást.)

A `next_day` függvény kap érték szerint paraméterként egy dátumot (azaz létrejön a függvényben egy lokális `struct date` változó, ami feltöltődik az aktuális paraméter értékével). A formális paraméternek megfelelő lokális változót megváltoztatjuk a `one_day_forward` függvény segítségével, majd a kapott struktúrát visszaadjuk. A kapott függvény használatát szemlélteni a `main`, amelyben a `sober` struktúrát a `next_day` függvény eredményével töltjük fel. A `new_year` struktúra eközben természetesen változatlan marad.

3.3 Unió típusok

Unió típus

Típusértékei több típus valamelyikéből

C: union

```
struct       month { char *name; int days; }; /* name and days */
union name_or_days { char *name; int days; }; /* either of them */

union name_or_days brrr = {"Pete"};  /* now it contains a name */
printf("%s\n", brrr.name);   /* fine */
printf("%d\n", brrr.days);   /* prints rubbish */
brrr.days = 42;              /* now it contains an int */
printf("%d\n", brrr.days);   /* fine */
printf("%s\n", brrr.name);   /* probably segmentation fault */
```

Az unió típusok arra valók, hogy több típus típusértékhalmazát egy típusba egyesítsék. Sok procedurális, imperatív nyelvben megjelent ez a fajta típuskonstrukció. Van néhány szép felhasználási módja, de egy komoly problémát is felvet: tudjuk-e, hogy az aktuális időpillanatban milyen típusú értéket tárol a változónk?

A C nyelvben a `union` kulcsszóval vezetünk be unió típust. A fenti, már jól ismert `struct month` struktúra két mezőt tartalmaz: egy `name` *és* egy `days` mezőt. A `union name_or_days` típusú változók azonban a kétfajta érték közül egyszerre mindig csak egyet tartalmaznak! Azt, hogy éppen melyiket, a programnak kell valahogy nyomon követnie. Amikor létrehozzuk a `brrr` változót, egy `char *` értéket teszünk bele, egy karaktermutatót a `"Pete"` sztringre. Hivatkozhatunk a `struct`-oknál megismert szelektorral erre a mezőre – de hivatkozhatunk a másik mezőre is: a C nyelvben sem fordítás közben, sem futás közben nem lehet kitalálni, milyen típusú értéket hordoz a változónk. A második `printf` ennek köszönhetően valami zagyvaságot ír ki. Meg is változtathatjuk a `brrr`-ben tárolt értéket, akár egy másik típusúra is. A harmadik kiírás rendben van, de a negyedik valószínűleg elszáll, mert a `42` számot próbáljuk karaktermutatóként értelmezni, és kiírni az „általa mutatott” sztringet.

Mi is történik? A `union` típusok belső ábrázolása olyan szokott lenni, hogy az összeuniózni kívánt típusok közül a legnagyobb helyigényűnek megfelelő tárhellyel dolgozik, és ezen a tárhelyen akármilyen értéket tárolhat az összeuniózott típusok akármelyikéből. Tehát a `name_or_days` típusú változóban vagy egy `char *` mutató, vagy egy `int` érték lesz; pontosabban az ott található értéket megpróbálhatjuk értelmezni így is, úgy is. A mi felelőségünk, hogy jól értelmezzük a unióban talált értéket.

3.4 Tömbök

Tömb fogalma

Azonos típusú (méretű) objektumok egymás után a memóriában.

- Bármelyik hatékonyan elérhető!
- Rögzített számú objektum!

```
int vector[4];
int matrix[5][3];   /* 15 elem sorfolytonosan */
```

Indexelés 0-tól

- `vector[i]` címe: `vector` címe + `i * sizeof(int)`
- `matrix[i][j]` címe: `matrix` címe + `(i * 3 + j) * sizeof(int)`

C tömbök deklarációja

```c
int a[4];                       /* 4 elemű, inicializálatlan */
int b[] = {1, 5, 2, 8};         /* 4 elemű */
int c[8] = {1, 5, 2, 8};        /* 8 elemű, 0-kkal feltöltve */
int d[3] = {1, 5, 2, 8};        /* 3 elemű, felesleg eldobva */

extern int e[];
extern int f[10];               /* méret ignorálva */

char s[] = "alma";
char z[] = {'a','l','m','a','\0'};

int m[5][3];                    /* 15 elem, sorfolytonosan */
int n[][3] = {{1,2,3},{2,3,4}}; /* méret nem elhagyható! */
int q[3][3][4][3];              /* 108 elem */
```

Tömbök indexelése

C

- `int t[] = {1,2,3,4};`
- 0-tól indexelünk
- hossz futás közben ismeretlen
- fordítás közben: `sizeof`
  - `sizeof(t) / sizeof(t[0])`
- hibás index: definiálatlanság

Python

- `t = [1,2,3,4]`
- 0-tól indexelünk
- hossz futás közben ismert
- negatív index is értelmezett
  - utolsó előtti elem: `t[-2]`
- hibás index: futási hiba

3.5 Mutatók

C mutatók

- Más változókra mutat(hat): indirekció
  - dinamikus
  - automatikus vagy statikus
- Típusbiztos

```c
int i;
int t[4];
int *p = NULL;  /* sehova sem mutat */
```

```c
/* dinamikus tárolású változóra mutat */
p = (int*)malloc( sizeof(int) * i ); ... free(p);

/* statikusra vagy automatikusra mutat */
p = &i;    p = t;

*p = 5;         /* dereferálás */
```

C deklarációk mutatókkal

```c
int i = 42;
int *p = &i;
int **pp = &p;          /* mutató mutatóra */
int *ps[10];            /* mutatók tömbje */
int (*pt)[10];          /* mutató tömbre */

char *str = "Hello!";

void *foo = str;        /* akármire mutathat */

int* p,q;               /* mutató és int */
int  s,t[5];            /* int és tömb */
int *f(void);           /* int* eredményű függvény */
int (*f)(void);         /* mutató int eredményű függvényre */
```

Tömbök és mutatók kapcsolata

- Tömb: *second-class citizen*
- Tömb → mutató
- Nem ekvivalensek!

```c
int t[] = {1,2,3};
t = {1,2,4};  /* fordítási hiba */

int *p = t;
int *q = &t[0];

int (*r)[3] = &t;

printf( "%d%d%d%d\n", t[0], *p, *q, (*r)[0] );
```

Érdekes módon a tömbök nem „teljes jogú állampolgárai” a C nyelvnek. Ez azt jelenti, hogy egy tömb típusú változót nem lehet olyan általánosan használni, mint mondjuk egy skalár típusú változót. Észrevehettük már, hogy a tömböknek nem lehet értéket adni. Ebből a szempontból a tömbök hasonlítanak a függvényekre. (Ha definiálunk egy `f` nevű függvényt, akkor innentől kezdve az `f` azonosító azt a függvényt fogja jelenteni, nem adhatunk neki „más értéket”.)

Sok esetben akkor, amikor tömböket használunk, valójában nem is tömböket használunk, hanem mutatókat. Ugyanis a tömb hivatkozások sokszor automatikusan mutatóvá konvertálódnak. Ez magyarázza meg azt is, hogy egy mutató típusú változónak miért lehet értékül adni egy tömb típusú változót. Valójában ez az értékadás azt jelenti, hogy a tömb legelső elemének a címét (ami egyben a tömb címe is!) kapja meg a mutató típusú változó.

A típusokkal kapcsolatban akadhat itt némi problémánk. A fenti példában az `r` változó egy mutató, ami egy 3 hosszú egész tömbre hivatkozhat. A `t` címével inicializáltuk, így az `r` a `t` tömb elejére mutat. Az értékét tekintve megegyezik tehát a `p` és a `q` mutatókkal, de az `r`-nek más a típusa. A `p = r` értékadás hibás, de a típuskényszerítéssel a típusok különbözősége áthidalható: `p = (int*)r`.

Ahogy látjuk, a tömbök és a mutatók néha ekvivalensnek tűnnek, de nem azok! Később látunk majd olyan példát, amelyből ez kiderül!

Tömb átadása paraméterként?

Valójában mutató típusú a paraméter!

```
double distance( double a[], double *b )
{
   double dx = a[0] - b[0],
          dy = a[1] - b[1],
          dz = a[2] - b[2];
   return sqrt(dx*dx + dy*dy + dz*dz);
}
```

Mutató-aritmetika – léptetések

```
int v[] = {6, 2, 8, 7, 3};
int *p = v;
int *q = v + 3;   /* v konvertálódik */
++p;
*p = 5;           /* v: 6, 5, 8, 7, 3 */
p += 2;
*q = 1;           /* v: 6, 5, 8, 1, 3 */
q -= 2;
*q = 2;           /* v: 6, 2, 8, 1, 3 */
```

Mutató-aritmetika – összehasonlítások

```
int v[] = {6, 2, 8, 7, 3};
int *p = v;
int *q = v + 3;

if ( p == q ) { ... }
if ( p != q ) { ... }
if ( p < q ) { ... }
if ( p <= q ) { ... }
if ( p > q ) { ... }
if ( p >= q ) { ... }
```

Mutató-aritmetika – indexelés

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

str[ 1 ] = 'o';
*( str + 1 ) = 'o';

printf( "%s\n", str + 3 );
printf( "%c\n", 3[ str ] );
```

Mutató-aritmetika: példa

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

Mutatók és tömbök közötti különbségek

```
int v[] = {6, 3, 7, 2};
int *p = v;

v[ 1 ] = 5;
p[ 1 ] = 8;

int w[] = {1,2,3};
p = w;  /* ok */
v = w;  /* fordítási hiba */

printf( "%d %d\n", sizeof( v ), sizeof( p ) );
```

Lásd még az utolsó példát itt: http://gsd.web.elte.hu/lectures/imper/imper-lecture-5/.

4 Tömbök átadása paraméterként

Tömbök átadása paraméterként: általánosítás?

```
double distance( double a[3], double b[3] ){
   double sum = 0.0;
   unsigned int i;
   for( i=0; i<3; ++i ){      /* beégetett érték :-( */
      double delta = a[i] - b[i];
      sum += delta*delta;
   }
   return sqrt( sum );
}

int main(){
   double p[3] = {36, 8, 3}, q[3] = {0, 0, 0};
   printf( "%f\n", distance(p,q) );
   return 0;
}
```

Itt a paraméterlistában a méretre adott megkötés elvész, amikor a tömb „átváltozik” mutatóvá. Inkább csak megtévesztő, hogy odaírtuk a tömbök méretét: semmilyen fordítási idejű vagy futási idejű ellenőrzést nem biztosít ezen információ megadása.

Tömbök paraméterként: fordítási időben rögzített méret

```
#define DIMENSION 3
```

```
double distance( double a[DIMENSION], double b[DIMENSION] ){
   double sum = 0.0;
   unsigned int i;
   for( i=0; i<DIMENSION; ++i ){
      double delta = a[i] - b[i];
      sum += delta*delta;
   }
   return sqrt( sum );
}

int main(){
   double p[DIMENSION] = {36, 8, 3}, q[DIMENSION] = {0, 0, 0};
   printf( "%f\n", distance(p,q) );
   return 0;
}
```

Tömbök paraméterként: futási időben rögzített méret?

```
double distance( double a[], double b[] ){
   double sum = 0.0;
   unsigned int i;
   for( i=0; i<???; ++i ){    /* ez itt nem Python */
      double delta = a[i] - b[i];
      sum += delta*delta;
   }
   return sqrt( sum );
}

int main(){
   double p[] = {3.0, 4.0}, q[] = {0.0, 0.0};
   printf( "%f\n", distance(p,q) );
   return 0;
}
```

Tömbök paraméterként: hibás megközelítés

```
double distance( double a[], double b[] ){
   double sum = 0.0;
   unsigned int i;
   for( i=0; i<sizeof(a)/sizeof(a[0]); ++i ){
      double delta = a[i] - b[i];
      sum += delta*delta;
   }
   return sqrt( sum );
}

int main(){
   double p[] = {3.0, 4.0}, q[] = {0.0, 0.0};
   printf( "%f\n", distance(p,q) );
   return 0;
}
```

Tömbök paraméterként: helyesen

```
double distance( double a[], double b[], int dim ){
   double sum = 0.0;
```

```
   unsigned int i;
   for( i=0; i<dim; ++i ){
      double delta = a[i] - b[i];
      sum += delta*delta;
   }
   return sqrt( sum );
}

int main(){
   double p[] = {3.0, 4.0}, q[] = {0.0, 0.0};
   printf( "%f\n", distance(p,q,sizeof(p)/sizeof(p[0])) );
   return 0;
}
```

Bonyolult struktúra átadása paraméterként

```
int main( int argc, char *argv[] ){ ... }
```

- `argc`: pozitív szám
- `argv[0]`: program neve
- `argv[i]`: parancssori argumentum ($1 \leq i <$ argc)
  - karaktertömb, végén `NUL` (`'\0'`)
- `argv[argc]`: `NULL`

```
int main( void ){ ... }
int main( int argc, char *argv[], char *envp[] ){ ... }
int main(){ ... }
```

Több dimenziós tömbök paraméterként

```
double m[4][4] = {{1,2,3,4},{1,2,3,4},{1,2,3,4},{1,2,3,4}};

transpose(m);

{
   int i,j;
   for( i=0; i<4; ++i ){
      for( j=0; j<4; ++j ){
         printf("%3.0f", m[i][j]);
      }
      printf("\n");
   }
}
```

Túl merev megoldás

```
void transpose( double matrix[4][4] ){  /* double matrix[][4] */
   int size = sizeof(matrix[0])/sizeof(matrix[0][0]);
   int i, j;
   for( i=1; i<size; ++i ){
      for( j=0; j<i; ++j ){
         double tmp = matrix[i][j];
         matrix[i][j] = matrix[j][i];
         matrix[j][i] = tmp;
      }
   }
}
```

```
double m[4][4] = {{1,2,3,4},{1,2,3,4},{1,2,3,4},{1,2,3,4}};
transpose(m);
```

Sorfolytonos ábrázolás: egybefüggő memóriaterület

```
void transpose( double *matrix, int size ){  /* size*size double */
   int i, j;
   for( i=1; i<size; ++i ){
      for( j=0; j<i; ++j ){
         int idx1 = i*size+j,  /* matrix[i][j] helyett */
             idx2 = j*size+i;  /* matrix[j][i] helyett */
         double tmp = matrix[idx1];
         matrix[idx1] = matrix[idx2];
         matrix[idx2] = tmp;
      }
   }
}

double m[4][4] = {{1,2,3,4},{1,2,3,4},{1,2,3,4},{1,2,3,4}};
transpose( &m[0][0], 4 );  /* transpose( (double*)m, 4 ) */
```

Alternatív reprezentáció: mutatók tömbje

```
void transpose( double *matrix[], int size ){
   int i, j;
   for( i=1; i<size; ++i ){
      for( j=0; j<i; ++j ){
         double tmp = matrix[i][j];
         matrix[i][j] = matrix[j][i];
         matrix[j][i] = tmp;
      }
   }
}

double m[4][4] = {{1,2,3,4},{1,2,3,4},{1,2,3,4},{1,2,3,4}};
double *helper[4]; for( i=0; i<4; ++i ) helper[i] = m[i];
transpose(helper,4);
```

5 Láncolt adatszerkezetek

Adatszerkezetek

- „Sok” adat szervezése
- Hatékony elérés, manipulálás
- Alapvető módszerek
  - Tömb alapú ábrázolás (indexelés)
  - Láncolt adatszerkezet
  - Hasítás

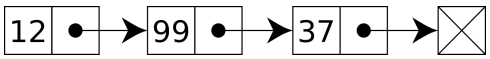
Az adatszerkezetek célja, hogy jó sok elemi adatot eltároljunk bennük, és ezeket utána kényelmesen és hatékonyan elérhessük, feldolgozhassuk. Nagyon sok féle adatszerkezetet kitaláltak már, a félév során ebből a tárgyból is láttunk párat. Van néhány alapvető fogás abban, hogy hogyan valósíthatjuk meg ezeket az adatszerkezeteket. Van olyan ábrázolás, amely a tömbök már jól ismert tulajdonságára épít: a tömb memóriabeli kezdőcíméből hatékonyan kiszámítható bármelyik adott indexű tömbelem memóriabeli

kezdőcíme (a tömbben tárolt elemek méretét felhasználva). Nem csak sorozatot, de például gráfot, fát (kupacot) is lehet tömb segítségével ábrázolni.

Mi most egy másfajta ábrázolással fogunk megismerkedni: a láncolt ábrázolással. Ennek lényegi eleme az, hogy az adatelemekre mutatókkal tudunk majd hivatkozni.

Láncolt adatszerkezetek

- Sorozat: láncolt lista
- Fa, pl. keresőfák
- Gráf



A láncolás során az adatokat olyan „csomópontokban'' helyezzük el, amelyeket mutatókkal kötünk össze. Ilyen technikával sorozatok könnyen leírhatók, ahogy az ábra mutatja. Ha egy csomópontban több mutató is van, akkor elágazó struktúrák, például fák is ábrázolhatók. Körkörös hivatkozásokkal akár tetszőleges gráfokat is felépíthetünk.

Sorozatok ábrázolása

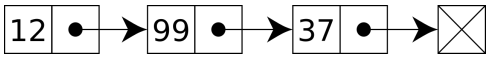
Tömb

- Akárhányadik elem előkeresése, felülírása
- Beszúrás/törlés?
  - Adatmozgatás
  - Újraallokálás

(egy példa: http://gsd.web.elte.hu/lectures/imper/imper-lecture-10/)

Láncolt lista

- Elemek előkeresése és felülírása bejárással
- Beszúrás/törlés bejárás során
- Akárhányadik elem előkeresése, felülírása?



Sorozatok ábrázolására két fő módszert szoktak használni: vagy egy tömbben helyezik el a sorozat elemeit, vagy egy láncolt listát építenek belőlük. A kétfajta reprezentációval más-más műveletek végezhetők el hatékonyan, ezért másfajta algoritmusok során szoktuk használni őket.

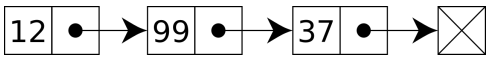
A tömb alapú reprezentáció az indexelésnek köszönhetően nagyon hatékony az elemek előkeresésében. Az adatszerkezetben tárol elemek számától függetlenül (úgy is mondjuk, hogy *konstans időben*), néhány aritmetikai művelettel meghatározható egy adott indexű elem memóriabeli helye. Így az adatelemek kiolvasása vagy felülírása könnyen elvégezhető.

A tömbök akkor használhatók jól, ha előre tudjuk, hogy hány elemet kívánunk használni az adatszerkezetben, vagy legalábbis tudunk egy jó felső korlátot mondani az elemek számára. Ha ugyanis „betelik" a tömb, a kibővítés elég költséges. Ilyen esetben új tömböt kell allokálni, és a régiből átmásolni az adatokat az újba. Ennek költsége a tömbben tárolt elemek számával egyenesen arányos (úgy is mondjuk, hogy *lineáris időben* végezhető el a művelet). A törlés a sorozatból szintén költséges. Ha nem szeretnénk lyukakat a reprezentációban, akkor egy törlés során a törölt elem után álló összes elemet eggyel előre kell mozgatni, ami ismét lineáris idejű művelet.

Ha olyan algoritmust készítünk, amelyben nagyon gyakran kell elemeket beszúrni a sorozatba, vagy törölni a sorozatból, meggondolandó, hogy nem hatékonyabb-e egy láncolt listás ábrázolás. A láncolt lista egy tetszőleges elemének előkeresése egy időigényes művelet, mert végig kell lépkedni a listán, amíg meg

nem találjuk. Ha viszont az algoritmusban erre nincs szükség, akkor a láncolt lista nagyon jól használható. Az elemek végigjárása (azaz a lista bejárása) hatékony: minden elem rákövetkezőjét konstans időben megkaphatjuk. Bejárás során a törlés és a beszúrás is konstans idejű.

Láncolt lista




```c
struct node
{
    int data;
    struct node *next;
};
```

```haskell
nums :: [Int]
nums = [12, 99, 37]
```

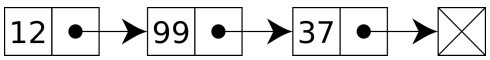
A C-ben a láncolt adatszerkezetek sokkal alacsonyabb szinten programozandók, mint mondjuk Haskellben. A láncolt lista nem egy nyelvbe beépített adatszerkezet, hanem mi magunk kell megvalósítsuk. Az ábrázoláshoz egy struktúrát érdemes használni. A struktúra egy csomópontot fog leírni a listában. Az egyik mezője a csomópontban tárolt adatelem (itt most `data`), a másik mezője pedig a lista következő csomópontjára mutat (`next`). A láncolást tehát mutatók segítségével fejezhetjük ki.

Most nézzük meg, hogyan építhető fel a fenti három elemű lista.

Láncolt lista felépítése

```c
struct node
{
    int data;
    struct node *next;
};

struct node *head;
head = (struct node *)malloc(sizeof(struct node));
head->data = 12;
head->next = (struct node *)malloc(sizeof(struct node));
head->next->data = 99;
head->next->next = (struct node *)malloc(sizeof(struct node));
head->next->next->data = 37;
head->next->next->next = NULL;
```



A lista csomópontjait dinamikusan hozzuk létre. Az allokált csomópontokat adatokkal töltjük fel, valamint a láncolást is elvégezzük a mutatók beállításával. Természetesen egy lista feltöltését általában egy ciklussal programozzuk le.

Beszúrás rendezett sorozatba (Haskell)

```haskell
insert (x:xs) y
    | x < y     = x:(insert xs y)
    | otherwise = y:x:xs
```

```
insert [] y = [y]
```

Tekintsük most azt a feladatot, hogy egy növekvően rendezett sorozatba be kell szúrnunk a megfelelő helyre egy újabb elemet. Ezt a feladatot Haskellben nagyon egyszerű megoldani. Nézzük most meg Pythonban is!

Beszúrás rendezett sorozatba (Python)

```
def insert(xs,y):
     if xs:
             z, *zs = xs
             if z < y:
                     return [z] + insert(zs,y)
             else:
                     return [y] + xs
     else:
             return [y]
```

Könnyen átírhattuk a rekurzív megoldást Pythonra is! Vegyük észre, hogy egy lista akkor "igaz", ha nem üres. Ezt használtuk ki az első elágazásban. Most alakítsuk át a Pythonos megoldásunkat úgy, hogy rekurzió helyett csak ciklust használunk!

Beszúrás rendezett sorozatba (Python, ciklussal)

```
def insert(xs,y):
    def pos():
        i = 0
        for v in xs:
            if v > y:
                return i
            else:
                i += 1
        return i
    xs.insert(pos(),y)
```

Ebben a megoldásban a listák `insert` műveletére támaszkodunk, mely segítségével a művelet elé ért listába a megadott pozícióra (első paraméter) bekerül a megadott elem (második paraméter). Kicsit kusza a kód abból a szempontból, hogy most mi is írtunk egy `insert` függvényt, meg a listáknak is van egy ilyen nevű művelete. A paraméterezésük különböző. Az általunk írt függvény paramerként kapja a listát (és a beszúrandó elemet), míg a listák beépített `insert` művelete egy úgynevezett *metódus*: a listát, amelyiken végrehajtjuk, a metódusnév és az aktuális paramerek előtt kell megadni, ponttal elválasztva a metódusnévtől.

A pozíciót az `insert` függvényre nézve lokális `pos` függvénnyel számoltuk ki, mely a számára non-lokális `xs` és `y` változókhoz hozzáfér.

Most tekintsünk egy C-nyelvű megvalósítást.

Beszúrás rendezett sorozatba (C)

```
struct node
{
    int data;
    struct node *next;
};

typedef struct node * list_t;
```

```
list_t insert( list_t list, int value );
```

Vezessünk be egy típusszinonímát a *mutató a* `node` *struktúrára* típushoz. Az `insert` függvény kap egy listát, amelybe beszúrja a kapott `value` értéket. A visszatérési értékben megkapjuk a módosított listát.

Beszúrás rendezett listába (C)

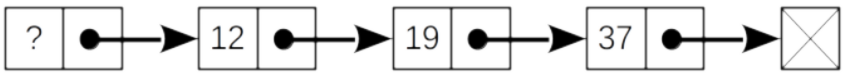
```
list_t insert( list_t list, int value ){
    list_t new = (list_t) malloc(sizeof(struct node));
    if( NULL == new ) return NULL;
    new->data = value;
    if( NULL == list || value < list->data ){
        new->next = list;
        return new;
    } else {
        list_t current = list, next = current->next;
        while( next != NULL && next->data < value ){
            current = next;
            next = current->next;
        }
        new->next = next;
        current->next = new;
        return list;
    }
}
```

A megvalósítás elég bonyolult. Egy ügyes trükkel egyszerűsíthetünk rajta. Ehhez bevezetjük a fejelemes lista fogalmát.

Egy másik módosítást az az elv motivál, hogy lehetőleg ugyanott szabadítsuk fel a dinamikus memóriát, ahol allokáltuk. Ezért a következő megoldásban a függvényünk már nem a beszúrandó értéket, hanem egy, a listába a megfelelő helyre beillesztendő `node` struktúrára hivatkozó mutatót kap.

Fejelemes lista

- Plusz egy `node` a lista legelején
- A benne tárolt értéket nem használjuk
- Ne kelljen az üres lista esettel külön foglalkozni



Beszúrás rendezett (fejelemes) listába

```
void insert( list_t list, struct node *new_node ){
    list_t current = list, next = current->next;
    while( next != NULL && next->data < new_node->data ){
        current = next;
        next = current->next;
    }
    new_node->next = next;
    current->next = new_node;
}
```

Ebben a megvalósításban megkapjuk a fejelemes rendezett listát (a fejelemre mutató mutatót), valamint egy mutatót egy `node` struktúrára. Ez a mutató mutat arra a csúcsra, amelyet be szeretnénk láncolni a megfelelő helyre a kapott rendezett listába.

Ezzel a technikával a `node` struktúrák dinamikus létrehozását ugyanarra a programrészre bízzuk, amelyik a felszabadítást is végzi majd. Például az alábbi kód ezen `insert` művelet segítségével lerendez egy egész számokból álló tömböt.

```
static void extract( int *data, list_t list ){   /* copy list to data */
    while( list != NULL ){
        *data = list->data;
        data++;
        list = list->next;
    }
}

int sort( int data[], int length ){
    list_t nodes = (list_t) malloc( (length+1) * sizeof(struct node) );
    list_t head;
    int i;
    if( NULL == nodes ) return 0;      // failure
    head = nodes+length;
    head->next = NULL;
    for( i=0; i<length; ++i){
        nodes[i].data = data[i];
        insert(head,nodes+i);
    }
    extract(data, head->next);
    free(nodes);
    return 1;                          // success
}
```

6 Egyenlőségvizsgálat és másolás

Egyenlőségvizsgálat és másolás elemi típusokon

```
int a = 5;
int b = 7;

if( a != b )
{
    a = b;
}
```

Mindenki kapásból megérti a fenti példakódot. Összehasonlítjuk a két `int` típusú változót, és ha nem egyenlőek, akkor az egyiknek értékül adjuk a másikat. Egyszerű. De mi történik, ha nem `int` típusú változókkal, hanem valami bonyolultabbal dolgozunk?

Mutatókkal?

```
int n = 4;
int *a = (int*)malloc(sizeof(int));
int *b = &n;

if( a != b )
{
    a = b;
}
```

A két mutató akkor egyenlő, ha ugyanoda mutatnak. Az értékadás hatására az `a` ugyanoda fog mutatni, mint a `b`, tehát `a` egyenlő lesz `b`-vel, és a `*a` ugyanaz lesz, mint a `*b`. Ezt neveztük aliasingnak.

Tömbökkel?

```
int a[] = {5,2};
int b[] = {5,2};

if( a != b )
```

```
{
    a = b;          /* fordítási hiba */
}
```

Ha ugyanezt a kódrészletet tömbökkel vizsgáljuk, azt látjuk, hogy az értékadás nem is lehetséges. Ezt már korábban is megállapítottuk: a C-ben tömböknek nem lehet új értéket adni. Az egyenlőségvizsgálat értelmezett, mégpedig úgy, hogy a tömbök automatikusan mutatóvá alakulnak, így két fizikailag különböző tömb nem lehet egyenlő egymással. (Az *egyenlőség* valójában az *azonosság*.)

Tömbökkel!

```
#define SIZE 3

int is_equal( int a[], int b[] ){
    for( int i=0; i<SIZE; ++i )
        if( a[i] != b[i] ) return 0;
    return 1;
}

void copy( int a[], int b[] ){
    for( int i=0; i<SIZE; ++i ) a[i] = b[i];
}

int a[SIZE] = {5,2}, b[SIZE] = {7,3,0};

...
if( ! is_equal(a,b) ) copy( a, b );
```

Ha tartalmi egyenlőségvizsgálatot szeretnénk a tömbjeinkhez, akkor érdemes lehet egy műveletet definiálni erre a célra. Ha fix méretű tömbjeink vannak (pl. `SIZE`), akkor ez elég könnyű, egyébként viszont külön kell vizsgálni, hogy a tömbök mérete megegyezik-e, és persze a közös méretet át is kell adni az egyenlőséget vizsgáló függvénynek. (Ne feledjük: nem a tömb, hanem a legelső elemére mutató mutató adódik át paraméterként!)

Mivel helyettesíthetjük az értékadást? Ha az a célunk, hogy a két tömb tartalmilag, elemről elemre megegyezzen, akkor írhatunk egy olyan másoló műveletet, ami az egyikből átmásolja a másikba az adatokat. Egyszerűbben is megúszhatjuk a dolgot: használhatjuk a `string` könyvtárban definiált `memcpy` függvényt is erre a célra.

```
void *memcpy( void *dest, const void *src, size_t numbytes );
```

A típusokból látható, hogy ezt akármilyen adatstruktúra másolására használhatjuk: a `dest` memóriacímtől kezdődően tárolja el azt a `numbytes` darab bájtot, amit az `src` memóriacímtől kezdődően kiolvas. Fontos megkötés erre a függvényre, hogy a két tárterület nem fedhet át egymással! A fenti `a` és `b` esetében ez teljesül, de amúgy nem feltétlen könnyű tetszőleges mutatóknál ezt garantálni…

```
#include <string.h>
#define MIN(a,b) ((a)>(b) ? (b) : (a))

int a[] = {5,2};
int b[] = {7,3,4};

memcpy( a, b, MIN(sizeof(a), sizeof(b)) );
```

Struktúrákkal?

```
struct pair { int x, y; };

struct pair a, b;
a.x = a.y = 1;
b.x = b.y = 2;

if( a != b )          /* fordítási hiba */
{
```

```
    a = b;
}
```

Térjünk át most struktúrákra. Rájuk meg az összehasonlítás tilos, arra kapunk fordítási hibát. Az értékadás értelmezett, és lényegében pont ugyanaz történik, mint elemi adatok esetén: a `b` változóban tárolt összetett érték átmásolódik az `a` változóba, így a két rekord mezői páronként egyenlően lesznek az értékadás után.

Struktúrákkal!

```
struct pair { int x, y; };

int is_equal( struct pair a, struct pair b )
{
    return (a.x == b.x) && (a.y == b.y);
}

struct pair a, b;
a.x = a.y = 1;
b.x = b.y = 2;

if( is_equal(a,b) )
{
    a = b;
}
```

A megoldás itt is az lehet, hogy írunk egy műveletet, amely a hiányzó nyelvi szolgáltatást megvalósítja, azaz egy olyan műveletet, amely a két struktúrát mezőről mezőre összehasonlítja.

A másoláshoz nem feltétlenül kell műveletet írni, hiszen az értékadás gondoskodik a tartalom átmásolásáról. Ha mégis külön műveletet szeretnénk írni, akkor nem ez a jó megoldás.

```
void copy( struct pair a, struct pair b )
{
    a = b;
}
...
copy(a,b);
```

Az érték szerinti paraméterátadás miatt egy ilyen műveletnek nem lenne semmiféle hatása a hívási környezetben. Helyette természetesen ezt írnánk.

```
void copy( struct pair *a, struct pair *b )
{
    *a = *b;
}
...
copy(&a,&b);
```

Láncolt lista?

```
struct node
{
    int data;
    struct node *next;
};

struct node *a, *b;
...

if( a != b )
{
    a = b;
}
```

Nézzük meg, mint mondhatunk egy láncolt listával megvalósított sorozat egyenlőségvizsgálatáról, illetve másolásáról. Az eddigiek alapján nyilvánvaló, hogy a fenti kód – bár teljesen helyes, értelmes – nem azt vizsgálja, hogy két sorozat tartalmilag megegyezik-e, illetve nem tartalmilag másolja az egyik sorozatot a másikba. Az `a == b` kifejezés az `a` és `b` mutatók egyenlőségét vizsgálja, azaz pontosan akkor igaz, ha `a` és `b` egymás aliasai (azaz ugyanarra a listára hivatkoznak). Az `a = b` értékadás hatása pedig az, hogy az `a` mutató ugyanarra a listára fog hivatkozni, mint a `b` mutató.

Sekély megoldás – nem jó ide

```
struct node
{
    int data;
    struct node *next;
};

int is_equal( struct node *a, struct node *b )
{
    return (a->data == b->data) && (a->next == b->next);
}

void copy( struct node *a, const struct node *b )
{
    *a = *b;
}
```

Ahogy a `struct pair` kapcsán megtettük, megírhatjuk a tartalmi egyenlőségvizsgálatot és másolást a `struct node` típusra is. Értelmezzük a kapott megoldást: két mutató, amelyek `struct node` típusú értékre mutatnak, nem csak akkor lesznek egyenlőek, ha ugyanoda mutatnak (azaz ugyanazt a listát hivatkozzák), hanem akkor is, ha nem azonos, de egyenlő struktúrákra mutatnak. Mit is jelent itt az egyenlőség? Azt, hogy mezőnként egyenlő a két mutatott struktúra: a két listában a legelső elemek megegyeznek, valamint a két listában ugyanaz a struktúra lesz az első csúcspont rákövetkezője. Valóban ezt akartuk? Nem! Még mindig túl szigorú feltétellel próbálunk dolgozni! Azt szeretnénk, ha azon túl, hogy a két listában a legelső elemek megegyeznek, a második csúcsponttól kezdődően ismét *egyenlőség* (és nem *azonosság*) állna fenn! Ezt a megközelítést *mély egyenlőségvizsgálatnak* (angolul *deep equality*) nevezzük, szemben azzal, amit ezen a dián látunk (melynek neve *sekély egyenlőségvizsgálat*, azaz *shallow equality*).

Ugyanilyen problémát figyelhetünk meg a `copy` művelet kapcsán is. A fenti megoldás egy *sekély másolást* (*shallow copy*) végez: belemásolja az `a` mutató által hivatkozott lista legelső csúcspontjába a `b` mutató által hivatkozott lista legelső csúcspontját, aminek az lesz a következménye, hogy a két lista valójában nem két független lista lesz, hanem a második csúcsponttól kezdve egybeesnek. A másolás esetében is a *mély másolás* (*deep copy*) lesz itt a jó megoldás.

Mély egyenlőségvizsgálat

```
struct node
{
    int data;
    struct node *next;
};

int is_equal( struct node *a, struct node *b )
{
    if( a == b ) return 1;
    if( (NULL == a) || (NULL == b) ) return 0;
    if( a->data != b->data ) return 0;
    return is_equal(a->next, b->next);
}
```

Az egyenlőségvizsgálatot könnyen megfogalmazhatjuk rekurzióval és esetszétválasztással. A fenti `is_equal` definíció ugyan rekurzív, de – mivel a rekurzív hívás a definíció legutolsó utasítása, azaz a definíció *végrekurzív* (*tail-recursive*), a fordító könnyen optimalizálja ciklussá.

Mély másolás

```
struct node {
    int data;
    struct node *next;
};

struct node *copy( const struct node *b ){
    if( NULL == b ) return NULL;
    struct node *a = (struct node*)malloc(sizeof(struct node));
    if( NULL != a ){
        a->data = b->data;
        a->next = copy(b->next);
    } /* else hibajelzés! */
    return a;
}
```

A mély másolást is könnyű rekurzívan definiálni. Hosszabb listák esetén azonban ez okozhat problémákat (megtelik a végrehajtási verem, vagy egyszerűen csak nagyon lassú lesz az implementáció). Ezért érdemes lehet ciklusra átírni.

```
struct node *copy( const struct node *b ){
    if( NULL == b ) return NULL;
    struct node *a = (struct node*)malloc(sizeof(struct node));
    if( NULL != a ){
        a->data = b->data;
        struct node *p = a;
        b = b->next;
        while( NULL != b )
        {
            p->next = (struct node*)malloc(sizeof(struct node));
            if( NULL != p->next )
            {
                p->next->data = b->data;
                p = p->next;
                b = b->next;
            } /* else hibajelzés! */
        }
    } /* else hibajelzés! */
    return a;
}
```

7 Magasabbrendű függvények

Magasabbrendű függvények

Python

```
def foreach(fun,list):
   for item in list: fun(item)

foreach( print,                [1,2,3,4,5] )
foreach( lambda v: print(v,v), [1,2,3,4,5] )
```

C: függvénymutatók segítségével

```
/* mutató int eredményű, paraméter nélküli függvényre */
int (*fp)(void);

/* int->int függvényt és int-et váró függvény int eredménnyel */
int twice( int (*f)(int), int n );
```

C: függvénymutatók

```
int twice( int (*f)(int), int n )
{
    n = (*f)(n);
    n = f(n);
    return n;
}

int inc( int n ){ return n+1; }


printf(  "%d\n",  twice( &inc, 5 )  );
```

Házi feladat: tömbök rendezése az `stdlib`-ben definiált `qsort` függvénnnyel.

C: függvénymutatók - néhány észrevétel

```
int inc( int n ){ return n+1; }

int (*f)(int) = &inc;
f = inc;
f(3) + (*f)(3);

int (*g)() = inc;
g(3,4);  g();
```

8 Polimorfizmus

Címkézett unió

```
enum shapes { CIRCLE, SQUARE, RECTANGLE };
struct    circle { double radius; };
struct    square { int side; };
struct rectangle { int a; int b; };

struct shape
{
    int x, y;
    enum shapes tag;
    union csr
    {
        struct    circle c;
        struct    square s;
        struct rectangle r;
    } variant;
};
```

A fent látható trükkel biztonságosabbá tehetjük az unió típus használatát. Körbevesszük az unió konstrukciót egy struktúrával, amelynek egyik mezőjét címkeként használjuk: ebből a címkéből fogjuk tudni, hogy milyen típusú értéket tárolunk egy változóban. Természetesen nagyon oda kell figyelnünk `struct shape` értékek konstruálásánál, hogy a címke összhangban legyen a `variant` tartalommal, és arra is oda kell figyelni, hogy nehogy később elrontsuk a tartalmat egy olyan értékadással, amely mondjuk a `variant` belső típusát megváltoztatja, de a címkét nem.

Vannak olyan programozási nyelvek, amelyek eleve csak a címkézett unió konstrukciót biztosítják a programozó számára. Az Ada nyelvben például nem lehet a belső reprezentációt elrontani: a nyelv futtató környezete ellenőrzi, hogy a változó típusú részeket a címkének megfelelően használjuk-e.

A Haskell nyelvben hasonlóképpen szigorú a címkézett unió konstrukció. Ebben a nyelvben az algebrai adattípusokkal fejezzük ki ugyanezt a gondolatot.

Egységes használat

```
struct shape
{
    int x, y;
    enum shapes tag;
    union csr
    {
        struct    circle c;
        struct    square s;
        struct rectangle r;
    } variant;
};

void move( struct shape *aShape, int dx, int dy ){
    aShape->x += dx;
    aShape->y += dy;
}
```

A `shape` címkézett unió típus lehetővé teszi azt, hogy egységesen használjuk a különböző típusú alakzatokat. Az eltolás műveletet általánosan megírhatjuk az összes alakzatra. Azt, hogy egy művelet többféle típusra is működik, *polimorfizmusnak* (többalakúságnak) nevezzük.

Használat esetszétválasztással

```
struct shape {
    int x, y;
    enum shapes tag;
    union csr {
        struct    circle c;
        struct    square s;
        struct rectangle r;
    } variant;
};

double leftmost( struct shape aShape ){
   switch( aShape.tag ){
       case CIRCLE: return aShape.x - aShape.variant.c.radius;
       default:     return aShape.x;
   }
}
```

Ha olyan műveletet készítünk, amely megvalósítása függ az alakzat milyenségétől, a művelet belsejében megvizsgálhatjuk a *tag*et. Így a polimorfizmus biztonságosan fenntartható.

Biztonságos létrehozás

```
struct shape {
    int x, y;
    enum shapes tag;
    union csr {
        struct    circle c;
        struct    square s;
        struct rectangle r;
    } variant;
};

struct shape make_circle( int cx, int cy, double radius ){
    struct shape c;
    c.x = cx; c.y = cy; c.tag = CIRCLE;
    c.variant.c.radius = radius;
```

```
    return c;
}
```

Bevezethetünk olyan függvényeket, amelyekkel a biztonságos létrehozás megkönnyíthető.

Ne felejtsük el azonban azt a tényt, hogy a struktúra belsejébe belelátunk, sőt, bármikor módosíthatjuk a belsejét. Ez a lehetőség továbbra is veszélyessé teszi a címkézett unió típusunk használatát. A belső ábrázolás elrejtése lenne a következő nagy feladat, de ez már egy nagy lépés lesz az objektum-orientált programozás irányába.

Az objektum-orientált progamozásban a fenti példához hasonló konstrukciók helyett az öröklődés mechanizmusát használjuk. Egy bázisosztály és a különböző leszármazottai segítségével a shape, a circle, a square és a rectangle típusok nagyon kényelmesen (és ami szintén fontos: később bővíthető módon) megfogalmazhatók.

A `switch`-nél rugalmasabb, bővíthetőbb megoldás?

```
double leftmost_default( struct shape *aShape ){
        return aShape->x;
}

double leftmost_in_circle( struct shape *aCircle ){
        return aCircle->x - aCircle->variant.c.radius;
}


...
```

Műveletek és adatok egy egységbe zárása

```
struct shape {
    int x, y;
    enum shapes tag;
    union csr {
        struct    circle c;
        struct    square s;
        struct rectangle r;
    } variant;
    double (*leftmost)( struct shape *this );  /* függvénymutató */
};

double leftmost_default( struct shape *aShape ){ ... }
double leftmost_in_circle( struct shape *aCircle ){ ... }

double leftmost( struct shape aShape ){
    return ashape.leftmost( &aShape );
}
```

Rugalmasan beállítható implementáció

```
struct shape {
    int x, y;
    ...
    double (*leftmost)( struct shape *this );
};

double leftmost_default( struct shape *aShape ){ ... }
double leftmost_in_circle( struct shape *aCircle ){ ... }

struct shape make_circle( int cx, int cy, double radius ){
    struct shape c;
    c.x = cx; c.y = cy; c.tag = CIRCLE;
    c.variant.c.radius = radius;
    c.leftmost = leftmost_in_circle;
```

```
    return c;
}
```

Osztály

- Objektum-orientált nyelvek
- Osztály: rekordszerű struktúra
  - Adattagok (mezők)
  - Műveletek (metódusok)
- Öröklődés: címkézett unió

Python osztály: rekord típus megvalósítására

```
class Month: pass

jan = Month()
jan.name, jan.days = 'January', 31
print(jan.name, '(with', jan.days, 'days)')
```

Az osztályokkal a következő félévben fogunk megismerkedni. Az osztály olyan rekordszerű konsrukció, amelybe becsomagoljuk az adatokon kívül a rajtuk végezhető alapvető műveleteket is. Az objektumorientált paradigma egy fontos eleme az osztályok közötti öröklődés kapcsolat, amely a címkézett unió típus egyfajta megvalósulása. (Annyiban más a címkézett uniótól, hogy bővíthető újabb típusokkal, nincs „beégetve” a definícióba az összes típus, amit össze kívánunk uniózni.)

A fenti Python kódban létrehozunk egy `Month` osztályt. Az osztály egy típus lesz, amelyből később egyedeket, úgynevezett objektumokat hozhatunk létre. Ilyen objektum a példában az, amit értékül adunk a `jan` változónak. A `jan` változó egy referencia, amely a `Month()` kifejezéssel létrehozot objektumra hivatkozik. A `Month` típusú objektum a dinamikus tárhelyen jön létre.

A kód pikantériája, hogy a `Month` osztályt teljesen üres struktúraként definiáltuk. (A `pass` utasítás az üres/skip utasítás a Pythonban, ez itt azt fejezi ki, hogy a `Month` osztályba nem került semmi.) Ennek megfelelően a `jan` változó által hivatkozott objektum kezdetben szintén üres, azaz nincs egy mezője (adattagja, Pythonos elnevezéssel: *attribútuma*) sem. Viszont a Python dinamikus természete megengedi, hogy egy üres objektumot a későbbiekben mezőkkel bővítsünk ki. Ez történik a következő sorban. Az értékadás létrehoz két mezőt az objektumban: a hónap nevét, illetve napjai számát. Miután a szükséges mezőket létrehoztuk, azokra a C-ből is ismert módon hivatkozhatunk.

Ezzel a megközelítéssel az pusztán a bajunk, hogy egy hónap létrehozásánál könnyű elfelejteni a mezőket is létrehozni. Ha a mezőket nem hozzuk létre, de megpróblálunk rájuk hivatkozni, akkor hibát kapunk, mint a következő példában.

```
feb = Month()
feb.name = 'February'
print(feb.name, '(with', feb.days, 'days)')   # hiba: nincs feb.days
```

Biztonságos inicializáció

```
class Month:
    def __init__(self,name,days):
        self.name, self.days = name, days

jan = Month( 'January', 31 )
print(jan.name, '(with', jan.days, 'days)')
feb = Month()  # hibás!
```

Most definiáltunk egy speciális jelentésű, `__init__` nevű műveletet az osztályunkban. A nevéhez híven ez a művelet lesz felelős a létrejövő objektumok inicializálásáért. Amikor egy objektumot létrehozunk (például a `jan` változónak történő értékadásban), muszáj megadnunk azokat a paramétereket, amelyeket az `__init__` elvár, ebben az esetben a `name` és a `days` értékét. A megfelelő mező létrehozásáért az `__init__` művelet felel, a `self.name` és `self.days` kifejezéseknek történő értékadással. Ebből az látható, hogy a `self` az `__init__` műveletben magát az inicializálni kívánt objektumot jelenti.

Műveletek

```
class Month:
    def __init__(self,name,days):
        self.name, self.days = name, days
    def isValid(self,day):
        return 0 < day <= self.days
    def __str__(self):
        return self.name + ' (with ' + str(self.days) + ' days)';

jan = Month( 'January', 31 )
print( jan.isValid(29) )
print(jan)
```

Az inicializáción és az általa (vagy később) létrehozott mezőkön kívül újabb műveleteket is definiálhatunk az osztályban. Például az `isValid` művelet segítségével eldönthető, hogy egy adott nap érvényes-e egy hónapban. Az `isValid` függvény első paramétere megint a vizsgált hónap objektumot jelenti, amelynek a `days` mezőjét összehasonlítjuk a kapott második paraméterrel, a `day` értékével. A művelet meghívásához a speciális (sok objektum-orientált nyelvre jellemző) szintaxist használjuk: `jan.isValid(29)`. Azaz a `self` paraméternek itt a `jan`, a `day` paraméternek a `29` aktuális paraméter lesz megfeleltetve. (Valójában ez ugyanaz, mintha azt írtuk volna, hogy `Month.isValid(jan,29)`.) Az, hogy az osztályba az adattagok (mezők) mellé becsomagoljuk a rajtuk értelemezett alapvető műveleteket (szokás ezeket *metódusnak hívni*), az objektum-orientált programozás egy alapvető tulajdonsága. Magát a technikát szokás *egységbe zárásnak* (enkapszuláció, encapsulation) nevezni, mely szerint: ami logikailag egybe tartozik, azt helyezzük el egy szintaktikus egységbe – jelen esetben az osztály definíciójába.

A harmadik művelet, amit a `Month` osztályban definiáltunk, az `__str__` nevet viseli. Korábban már szót ejtettünk arról, hogy a Pythonban a dupla aláhúzással kezdődő és végződő függvénynevek speciális jelentést szoktak hordozni (pl. `__main__`). Ezt az ismeretet támasztotta alá az `__init__` is, melynek a specialitása abban áll, hogy objektum létrehozásánál ez kell, hogy meghívódjon. (Az objektumok létrehozásánál meghívandó műveleteket szokás *konstruktornak* is nevezni.) Ebbol már gyaníthatjuk, hogy az `__str__` metódus is speciális jelentéssel bír. Valóban, amikor egy kifejezésre meghívjuk az `str` függvényt, akkor a kifejezés által meghatározott érték `__str__` metódusa fog végrehajtódni. Ez történik akkor is, amikor a `print` művelettel kiírjuk a hónap objektumunkat a képernyőre: a `print` meghívja a `jan`-ra az `str` függvényt, ami sztringgé kell, hogy alakítsa a hónapot. Az `str` függvény pedig az általunk megadott `__str__` metódust hajtja végre.

```
jan.__str__()
```

Ha nem írjuk meg a `Month` osztályhoz az `__str__` metódust, akkor egy alapértelmezett megvalósítás hajtódik végre, amely a referencia értékét adja vissza sztring formájában. Azt, hogy egy osztályban megváltoztatjuk a megvalósítását egy műveletnek a korábbihoz (megörökölthöz vagy alapértelmezetthez) képest, *felüldefiniálásnak* (override, redefine) szokás nevezni – ez is egy alapvető lehetősége az objektum-orientált programozásnak.

C++ osztály

```
#include <string>

struct Month
{
    std::string name;
    int days;
    bool isValid( int day ) const
    {
        return 0 < day && day <= days;
    }
};
```

C++ objektum

```
#include <iostream>
#include <string>
```

```
struct Month {
    std::string name;
    int days;
    bool isValid(int day) const { return 0 < day && day <= days; }
};

int main() {
    Month jan;
    jan.name = "January";
    jan.days = 31;
    std::cout << jan.name << " (with " << jan.days << " days)"
              << std::endl;
}
```

C++ konstruktor

```
struct Month {
    std::string name;
    int days;
    bool isValid(int day) const { return 0 < day && day <= days; }
    Month(std::string name, int days) {
        this->name = name;
        this->days = days;
    }
};

int main() {
    Month jan("January",31);
    std::cout << jan.name << " (with " << jan.days << " days)"
              << std::endl;
}
```

Információelrejtés

```
class Month
{
public:
    bool isValid(int day) const { return 0 < day && day <= days; }
    Month(std::string name, int days)
    {
        this->name = name;
        this->days = days;
    }
    std::string getname() const { return this->name; }
    int getdays() const { return this->days; }
private:
    std::string name;
    int days;
};

int main()
{
    Month jan("January",31);
    std::cout << jan.getname() << " (with " << jan.getdays() << " days)"
             << std::endl;
}
```

Külön fordítás

Month.h

```
#ifndef MONTH_H
#define MONTH_H
#include <string>
class Month
{
public:
    bool isValid( int day ) const;
    Month(std::string name, int days);
    std::string getname() const;
    int getdays() const;
private:
    std::string name;
    int days;
};
#endif
```

Külön fordítás

Month.cpp

```
#include "Month.h"

bool Month::isValid(int day) const
{
    return 0 < day && day <= days;
}

Month::Month(std::string name, int days)
{
    this->name = name;
    this->days = days;
}

std::string Month::getname() const { return this->name; }
int Month::getdays() const { return this->days; }
```

Külön fordítás

main.cpp

```
#include "Month.h"
#include <iostream>

int main()
{
    Month jan("January",31);
    std::cout << jan.getname()
              << " (with "
              << jan.getdays()
              << " days)"
              << std::endl;
}
```

Névterek

```
#ifndef MONTH_H
#define MONTH_H
#include <string>
namespace ktolib {
    class Month {
        public:
```

```
            bool isValid( int day ) const;
            Month(std::string name, int days);
            std::string getname() const;
            int getdays() const;
        private:
            std::string name;
            int days;
    };
}
#endif
```