Imperatív programozás
Dinamikus memóriakezelés

Kozsik Tamás és mások

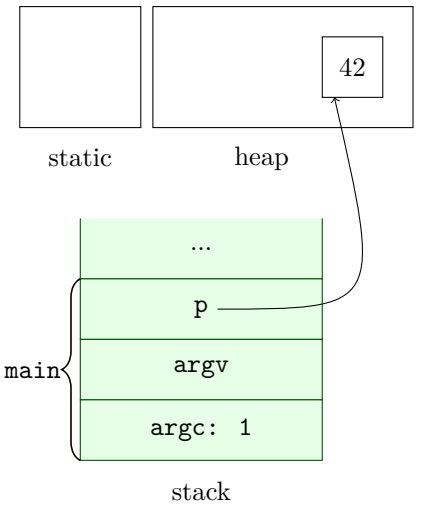
Dinamikus memóriakezelés

- Dinamikus tárolású „változók”
  - Heap (dinamikus tárhely)
- Élettartam: programozható
  - Létrehozás: allokáló utasítással
  - Felszabadítás
    - Felszabadító utasítás (C)
    - Szemétgyűjtés – garbage collection (Python)
- Használat: indirekció
  - Mutató – pointer (C)
  - Referencia – reference (Python)

Mutatók C-ben

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

int main()
{
    int *p;
    p = (int*)malloc(sizeof(int));
    if( NULL != p )
    {
        *p = 42;
        printf("%d\n", *p);
        free(p);
        return 0;
    }
    else return 1;
}
```



Összetevők

- Mutató (típusú) változó: `int *p;`
  - Vigyázat: `int* p, v;`
  - Hasonlóan: `int v, t[10];`
- Dereferálás (hova mutat?): `*p`
- „Sehova sem mutat”: `NULL`
- Allokálás és felszabadítás: `malloc` és `free` (`stdlib.h`)
  - Típuskényszerítés: `void*` → pl. `int*`

Mire jó?

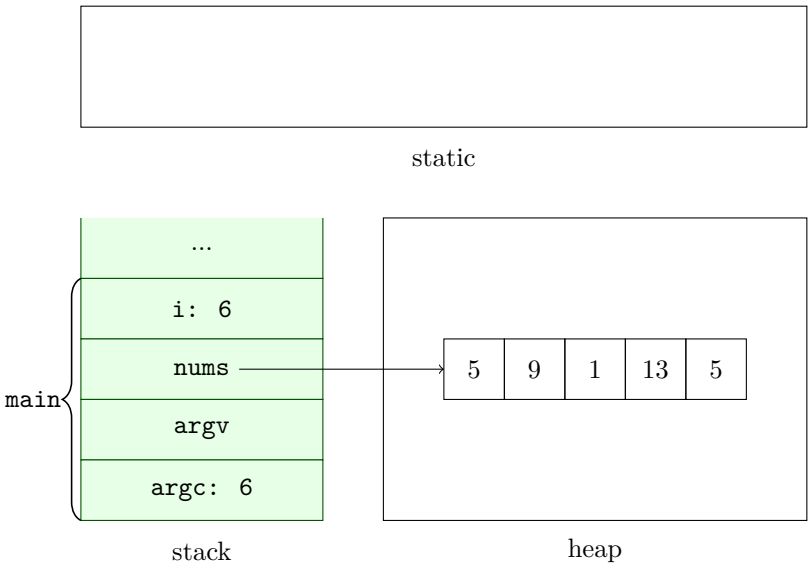
- Dinamikus méretű adat(-struktúra)
- Láncolt adatszerkezet
- Kimenő szemantikájú paraméterátadás
- …

Dinamikus méretű adatszerkezet

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

int main( int argc, char* argv[] ){
    int *nums = (int*)malloc((argc-1)*sizeof(int));
    if( NULL != nums ){
        int i;
        for( i=1; i<argc; ++i ) nums[i-1] = atoi(argv[i]);
        /* TODO: sort nums */
        for( i=1; i<argc; ++i ) printf("%d\n", nums[i-1]);
        free(nums);
        return 0;
    } else return 1;
}
```

Dinamikus méretű adatszerkezet



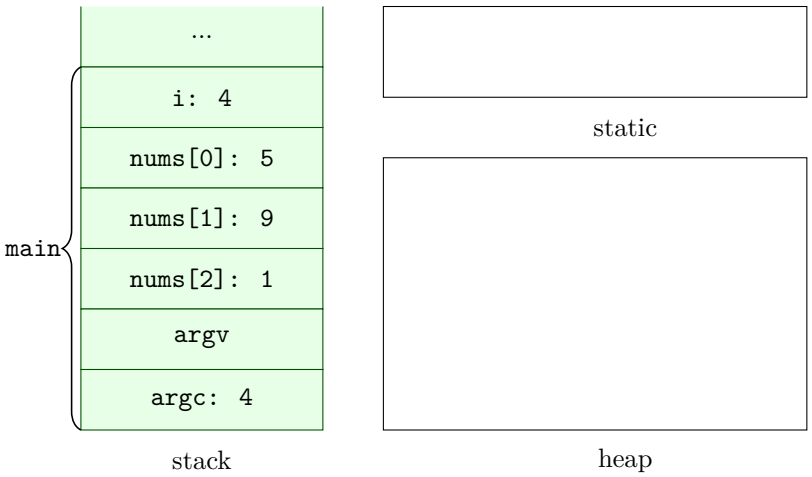
Kerülendő megoldás

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

int main( int argc, char* argv[] ){
    int nums[argc-1];
    int i;
    for( i=1; i<argc; ++i ) nums[i-1] = atoi(argv[i]);
    /* TODO: sort nums */
    for( i=1; i<argc; ++i ) printf("%d\n", nums[i-1]);
    return 0;
}
```

- C99: Variable Length Array (VLA)
- Nincs az ANSI C és C++ szabványokban

Kerülendő: VLA



Láncolt adatszerkezet

- Sorozat típus
- Bináris fa típus

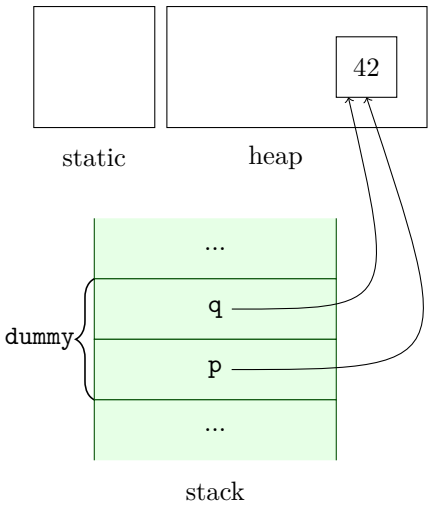
- Gráf típus
- ...

Bejárás közben konstans idejű törlés/beszúrás

Aliasing

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

void dummy(void)
{
    int *p, *q;
    p = (int*)malloc(sizeof(int));
    if( NULL != p ){
        q = p;
        *p = 42;
        printf("%d\n", *q);
        free(p);
    }
}
```



Felszabadítás

Minden dinamikusan létrehozott változót pontosan egyszer!

- Ha többször próbálom: hiba
- Ha egyszer sem: „elszivárog a memória” (memory leak)

Felszabadított változóra hivatkozni hiba!

Hivatkozás felszabadított változóra

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

void dummy(void)
{
    int *p, *q;
    p = (int*)malloc(sizeof(int));
    if( NULL != p ){
        q = p;
        *p = 42;
        free(p);
```

```
        printf("%d\n", *q);   /* hiba */
    }
}
```

Többszörösen felszabadított változó

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

void dummy(void)
{
    int *p, *q;
    p = (int*)malloc(sizeof(int));
    if( NULL != p ){
        q = p;
        *p = 42;
        printf("%d\n", *q);
        free(p);
        free(q);      /* hiba */
    }
}
```

Fel nem szabadított változó

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

void dummy(void)
{
    int *p, *q;
    p = (int*)malloc(sizeof(int));
    if( NULL != p ){
        q = p;
        *p = 42;
        printf("%d\n", *q);
    }      /* hiba */
}
```

Tulajdonos?

```
void dummy(void)
{
    int *q;
    {
        int *p = (int*)malloc(sizeof(int));
        q = p;
        if( NULL != p ){
            *p = 42;
        }
    }
    if( NULL != q ){
        printf("%d\n", *q);
        free(q);
    }
}
```

Könnyű elrontani!

```
int *produce( int argc, char* argv[] ){
    int *nums = (int*)malloc((argc-1)*sizeof(int));
```

```
    if( NULL != nums ){
        for( int i=1; i<argc; ++i ) nums[i-1] = atoi(argv[i]);
    }
    return nums;
}
void consume( int n, int *nums ){
    for( int i=0; i<n; ++i ) printf("%d\n", nums[i]);
    free(nums);
}
int main( int argc, char* argv[] ){
    int *nums = produce(argc,argv);
    if( NULL != nums ){ /* TODO: sort nums */ consume(argc-1,nums); }
    return (NULL == nums);
}
```

Alias C-ben és Pythonban

Alias: ugyanarra a tárterületre többféle névvel hivatkozhatunk

C: mutatók

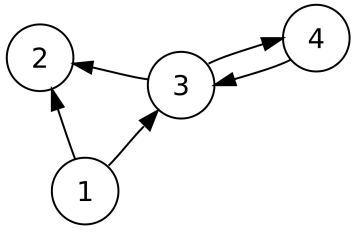
```
int xs[] = {1,2,3};
int *ys = xs;
xs[2] = 4;
printf("%d\n", ys[2]);
```

Python: referenciák

```
xs = [1,2,3]
ys = xs
xs[2] = 4
print(ys[2])
```

Amikor ugyanazt a tárterületet többféle névvel (változóval, kifejezéssel) is elérhetjük, akkor azt mondjuk, hogy ezek a nevek egymás álnevei, *alias*ai. A fenti példában az *ikszek* és az *ipszilonok* (`xs` és `ys`) ugyanazt a számsorozatot jelölik. Amikor értékül adjuk az `ys`-nek az `xs`-t, akkor aliast hozunk létre. Az aliasing nagyon be tudja csapni az embert: ha egy név segítségével megváltoztatjuk a tárhelyen tárolt értéket, akkor ez a változás a másik néven keresztül is megfigyelhetővé válik. Ezért ír ki a fenti példa mind a C, mind a Python esetében 4-et. (A mutatók és a tömbök kapcsolata C-ben elég érdekes, erre később még visszatérünk.)

Az aliasing jelenséggel találkozhatunk akkor, amikor dinamikus tárolású változókat kezelünk, mint ebben a példában, de máskor is (például cím szerinti paraméterátadásnál). Könnyű az alias miatt hibát véteni, de maga a jelenség nagyon hasznos is tud lenni. Például, ha irányított gráfokat szeretnénk a programunkban ábrázolni, akkor a gráf éleit hivatkozásokkal (mutatókkal, referenciákkal) adhatjuk meg. Az aliasing ebben az esetben annak felel meg, hogy a gráf egy csúcsába több más csúcsból vezet él (azaz a csúcs be-foka nagyobb, mint 1).



Mutató gyűjtőtípusa

```
int *p = (int *)malloc(sizeof(int));
if( NULL != p )
{
    *p = 123;
```

```
    *p = 12.3;  /* automatikus típuskonverzió */
    printf("%d\n", *p);
    free(p);
}
```

Érdekes különbség a C és a Python között az, hogy hogyan hivatkozunk a dinamikus tárolású változókra. A C-ben egy mutató típusú változót használhatunk erre a célra. Ennek a változónak a típusa állandó, mindig csak ugyanolyan típusú adatra hivatkozhat. Ez a statikus típusrendszer következménye. Az `int *p` deklaráció után a `p` egy olyan mutató lesz, amellyel csak `int` típusú tárterületre hivatkozhatunk. Ezt úgy is szoktuk mondani, hogy a `p` *gyűjtőtípusa* az `int`. A fenti példában a `12.3` érték automatikusan konvertálódik `int` típusúra, amikor értékül adjuk a `*p`-nek, mert a fordító tudja, hogy a `*p` típusa `int`.

Mutató gyűjtőtípusa: típuskényszerítés

```
float *q = (float *)malloc(sizeof(float));
if( NULL != q )
{
    int *p = (int *)q;
    *q = 12.3;
    printf("%d\n",*p);
    free(q);
}
```

Kivételt jelent a fenti szabály alól, ha típuskényszerítést, *type cast*ot alkalmazunk. Ez erőszakkal éri el azt, amit a statikus típusrendszer egyébként nem enged meg. Például egy `float *q` változó hivatkozhat ugyanarra a tárterületre, mint a `p`, ha ezt az értékadást végrehajtjuk: `p = (int *)q`. Ha a `*q` hivatkozáson keresztül beállítunk egy `float` értéket a dinamikus változóba, akkor a `*p` hivatkozáson keresztül egy „fura” értéket fogunk látni. Épp emiatt a fura érték miatt várja el a fordító, hogy a típuskényszerítés használatával megerősítsük a döntésünket, miszerint a `p` és a `q` egymás aliasa legyen.

Dinamikus tárhely elérése

C

- Explicit (mutató)
- Statikus típusellenőrzés
- Erősen típusos
- Felszabadítás

Python

- Implicit
- Dinamikus típusellenőrzés
- Erősen típusos
- Szemétgyűjtés

A C-vel szemben a Python nyelv nem jelzi explixit módon a mutatókat. **Valójában minden érték dinamikusan tárolódik, és a változók pusztán hivatkozások ezekre az értékekre.** Ez teszi lehetővé azt, hogy a változók típusa futás közben megváltozhasson.

```
v = 42       # egész
v = [1,2,3]  # lista
```

A dinamikus típusellenőrzés azt jelenti, hogy a program futása során minden művelet végrehajtása előtt ellenőrizzük, hogy a művelethez használt értékek típusa megfelelő-e. Az erős típusosság gondoskodik arról, hogy ne hajtsuk végre a műveleteket hibás (típusú) értékekkel, de nem statikusan, fordítás során, hanem dinamikusan, futás közben történik az ellenőrzés. Ezért nincs értelme gyűjtőtípusról beszélni. A változók pillanatnyi típusát az határozza meg, hogy épp akkor milyen típusú értékre hivatkoznak.

Egy másik fontos, már az előző előadáson érintett különbség a két nyelv között, hogy a már nem használt dinamikus változókat a C-ben explicit felszabadítjuk (`free`), míg a Python maga gondoskodik a feleslegessé (elérhetetlenné) vált dinamikus változók megszüntetéséről.

A `del` utasítás

C: dinamikus változó felszabadítása

```c
int *p = ...
int *q = p;
free(p);
printf("%d",*q);
```

Python: hivatkozás törlése

```python
v = [1,2,3]
u = v
del v   # v becomes undefined
print(u)
```

A Pythonban létezik egy `del` utasítás, amelyet jó nem összetéveszteni a C-beli `free()`-vel. A fenti C kódban a kiíró utasítás értelmetlen, mert egy felszabadított tárhely tartalmát probálja kiírni: a `free(p)` felszabadította a dinamikus változót. Ezzel szemben a Python kód remekül működik. A `del` utasítás a `v` változót, azaz a hivatkozást szünteti meg, nem a hivatkozott dinamikus változót. Emiatt az `u` változó, amely a `del` előtt az aliasa volt a `v`-nek, továbbra is egy érvényes értékre hivatkozik.

A `del v` után a `v` változó nem definiált. Ez nem ugyanaz, mintha `None` értékre (C-ben ennek leginkább a `NULL` mutató felel meg) állítottuk volna.

„Módosítható” és „nem módosítható” típusok

Mutable: list

```python
v = [1,2,3]
print(v[2]) # 3
v[2] = 4
print(v)
```

Immutable: tuple

```python
v = (1,2,3)
print(v[2]) # 3
v[2] = 4  # TypeError: 'tuple' object
   # does not support item assignment
```

A referenciák kezelése Pythonban teljesen transzparens. Ami viszont érdekes, az az, hogy bizonyos értékek megváltoztathatók (mutable), míg más értékek létrehozás után megváltoztathatatlanok (immutable). Egy közismert példát mutat be a fenti két kódrészlet. A listák és a rendezett n-esek (tuple) seítségével sorozatokat írhatunk le Pythonban. Egy fontos különbség kettejük között, hogy a listák megváltoztathatók, a tuple-ök pedig nem. Mindkét példában a `v` változó egy referencia, de amikor tuple értékre hivatkozik, a harmadik elem megváltoztatása sikertelen.

„Módosító” értékadás

Mutable: list

```python
v = [1,2,3]
u = v
v += [4,5]
print(v)    # [1,2,3,4,5]
print(u)    # [1,2,3,4,5]
```

Immutable: tuple

```python
v = (1,2,3)
u = v
v += (4,5)  # v = v + (4,5)
```

```
print(v)    # (1,2,3,4,5)
print(u)    # (1,2,3)
```

A módosíthatatlanságnak köszönhetően a „módosító” értékadások is másképp értelmezettek a listákra és a tuple-ökre. A `+=` a listák esetén a lista megváltoztatásával jár: a meglévő listához hozzáfűzzük a jobboldali operandust. Az aliason keresztül is a megváltozott listát látjuk.

A tuple esetében más a helyzet. Kezdetben az `u` és a `v` ugyanarra a tuple-re hivatkozik, de a `+=` ebben az esetben azt jelenti, hogy létrehozunk egy új tuple-t a régi `(1,2,3)` és a hozzáfűzendő `(4,5)` segítségével. A `v` hivatkozást erre az új tuple-re irányítjuk, míg az `u` hivatkozás marad a korábbi tuple-ön.

Mi okozza ezt a kettősséget? Az, hogy a listáknak van egy `__iadd__` nevű műveletük (metódusuk, ahogy az objektum-orientált programozásban hívjuk), és a Python ezt hívja meg az első példában, amikor a `+=` operátort használjuk. A listák `__iadd__` művelete úgy van elkészítve, hogy a listát megváltoztassa. A tuple-ök viszont módosíthatatlanok, így tartalmat megváltoztató műveleteket nem definiál(hat)tak hozzá. Nem is rendelkezik `__iadd__` nevű művelettel. Amikor a Python a második példában a `+=` operátorral találkozik, észleli, hogy az `__iadd__` itt nem áll rendelkezésre, ezért a `v += (4,5)` értékadást `v = v + (4,5)` formában értelmezi.

Beépített Python típusok

Mutable

- list, pl. `[1,2,3]`
- set, pl. `{1,2,3}`
- dictionary, pl. `{'a':1, 'b':2, 1:'a'}`

Immutable

- tuple, pl. `(1,2,3)`
- frozenset, pl. `frozenset({1,2,3})`
- range, pl. `range(1,23)`
- numeric: int, float, complex
- text, pl. `'123'`

Van néhány nagyon kényelmesen használható, beépített típus a Python nyelvben. Természetesen érdemes tudni, hogy melyik mutable, és melyik immutable. Ki is kísérletezhetjük.

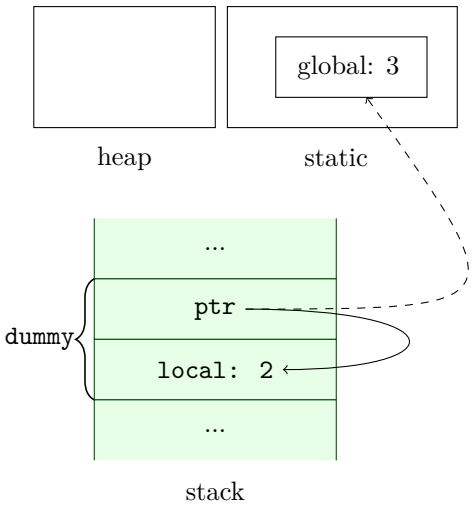
```
s = {1,2,3}
d = {'a':1, 'b':2, 1:'a'}
fs = frozenset({s})
t = '123'

s |= {4,5}
fs = fs | {4,5}
t = t + '45'
d.update({'c':3, 2:'b', 3:'c'})
```

Mutató nem dinamikus változóra

```
int global = 1;

void dummy(void)
{
   int local = 2;
   int *ptr;
   ptr = &global; *ptr = 3;
   ptr = &local;  *ptr = 4;
}
```



Érvénytelen mutató

Értelmetlen

```
int *make_ptr(void)
{

    int n = 42;
    return &n;
}
```

Értelmes

```
int *make_ptr(void)
{
    int *ptr = (int*)malloc(sizeof(int));
    *ptr = 42;
    return ptr;
}
```

```
        printf("%d\n", *make_ptr());
```

```
int *make_ptr(void)
{
    int *ptr = (int*)malloc(sizeof(int));
    *ptr = 42;
    return ptr;
}

int main(){
    int *ptr = make_ptr();
    if( NULL != ptr ){
        printf("%d\n", *make_ptr());
        free(ptr);
        return 0;
    } else return 1;
}
```