

## Erőforrások kezelése: a dinamikus memória és a file

A jegyzetben és a feladatok során két erőforrást ismerünk meg és használunk: a dinamikus memóriát és a file-t. Azonban minden erőforrás – például portok (usb, printer, ethernet, ...) – kezelésére hasonló általános kijelentések mondhatóak és hasonlóan kell őket is kezelni. Ezek az általános szabályok:

- kérni kell a hozzáférést
- a hozzáférés nem garantált, ezt ellenőrizni szükséges
  - siker esetén használhatjuk
  - a sikertelen kérést kezelniük kell
- az erőforrást fel kell szabadítani

Az elsőt nyilvánvaló okok miatt nem szokás elfelejteni, azonban a másik két teendő elfelejtődik, pedig a hozzáférés sikertelensége esetén jogosulatlan erőforrást használnánk (memória esetén segmentation fault hiba), a felszabadítás elmaradásakor pedig feleslegesen fogjuk az erőforrást, akár a program befejezése után is (memória esetén ezt memóriaszivárgásnak vagy angolul memory leaknek nevezzük). Az erőforrások felszabadítását meg kell tennünk akkor is, ha a program hibára fut.

### Dinamikus memória

Fizikailag a memória egy és oszthatatlan. A *stack*, amit ezidáig használtunk, a memória egyik felén található, - a legtöbb architektúrán - a legnagyobb memóriacímről indítva lefelé haladva. Ezen lépdel a processzor és hajtja végre a tárolt utasításokat vagy a tárolt adatokat olvassa. Minden függvényhívásnál a stack tetejére kerül a függvény, argumentumaival együtt és a függvényen belüli utasításokkal. Mikor visszatér a függvény, úgy a hívó utasítás memóriacímére ugrik vissza a processzor futása a *stack*-n, ennek következményeként pedig a magasabb (alacsonyabb) memórián lévő függvényünk minden lokálisa elveszik (törlés nem történik, ezért célszerű inicializálni egy változót, hogy ne az ott található memóriaszemét – értsd: más utasítások, adatok emlékei – legyen tárolva az aktuális változóban).

A memória másik feléről indítva ugyanígy foglalható. Ekkor a foglalt terület tetszőleges ideig a rendelkezésünkre áll, egészen annak felszabadításáig. Az ilyen memória több függvényen keresztül is használható.

A három feladatkört egyesével vizsgáljuk meg. Ezen műveletekhez a C nyelv standard *stdlib.h* könyvtárában implementált függvények állnak a rendelkezésünkre.

### Hozzáférés kérése:

- `void * malloc(size_t size)`  
A megadott *size* méretű memóriát próbálja lefoglalni byte-ban mérve (a *size\_t* architektúra és fordító függően valamilyen *unsigned* egész típus). Sikeres foglalás esetén visszatér egy általános *void* mutatóval a memória első címére, ellenkező esetben 0-val.
- `void * calloc(size_t num, size_t size)`  
Ritkábban használt allokálor függvény. A *num* számú és *size* méretű objektumok foglalására lehet használni, a foglalt memória nem feltétlenül a két szám szorzata. Sikeres foglalás esetén visszatér egy általános *void* mutatóval a memória első címére, ellenkező esetben 0-val. A sikeresen foglalt memória minden bitje nullára lesz inicializálva.

- `void* realloc(void * original, size_t new_size)`  
Az *original* mutató által hivatkozott *heap*-n található, már lefoglalt (és nem felszabadított) memóriaterületet allokalja át egy másik memóriaterületre, a *size* által meghatározott mérettel.
  - Sikeres foglalás esetén:
    - visszatér egy általános *void* mutatóval a memória első címére
    - a régi memóriaterület felszabadul
    - a régi memóriaterületen lévő adat átmásolódik az új területre
      - ha a *new\_size* nagyobb, mint az eredeti, úgy az eredeti méret feletti memóriában memóriaszemét lesz (nincs inicializálva)
      - ha a *new\_size* kisebb, mint az eredeti, úgy csak az ekkora méretnek megfelelő adat kerül másolásra, a többi elveszik
  - Sikertelen foglalás esetén:
    - null pointerrel tér vissza
    - a régi memóriaterület megmarad

Általános típusú adat nincs, azaz ha szeretnénk dolgozni a foglalt memóriával, konvertálni kell a megfelelő típusra. Ez megtehető impliciten is. Például:

```
int * data = malloc(10 * sizeof(int)); // 10 méretű int tömb foglalása
```

ahol egy (*int\**) implicit cast történik foglalás után.

### Ellenőrzés:

Mint a fenti függvények leírásánál láttuk, mindegyik allokaló függvény null pointerrel (szimplán 0-val) tér vissza, ha sikertelen a kérés. Ezt könnyen felhasználhatjuk a kérés sikertelenségének kezeléséhez:

```
if(data == 0) // ekvivalens ezzel: !data
{
    // sikertelen foglalás kezelése
}
```

Sikertelen foglalást számos módon kezelhetünk, például újra megkíséreljük a foglalást, hibaüzenetet írunk ki vagy termináljuk a programot. A null pointeren keresztül természetesen nem érhető el semmi, felhasználása segmentation fault-t dob.

### Felszabadítás:

Bármely módszerrel is foglalunk vagy újfoglalunk memóriát sikeresen, annak felszabadítása a programozó feladata, melyet a következő standard könyvtári függvénnyel oldhatunk meg.

```
void free(void * pointer)
```

Paraméterként a felszabadítandó területre mutató pointer használjuk. A felszabadítás mindig sikeres. A felszabadított memóriát a továbbiakban nem használhatjuk, a jól ismert segmentation fault hibát eredményezi, ha mégis megteesszük. Célszerű a pointert 0-ra állítani a felszabadítás után.

### Hibák:

- *memory leak*

Amennyiben nem szabadítjuk fel a foglalt memóriát, úgy memóriaszivárgás történik. Tipikus példája, amikor a *free* utasítást meg sem adjuk, de rejtettebb módon is elkövethetjük ezt a hibát, amikor a *pointer*t, ami a foglalt memóriára hivatkozik, egyszerűen átirányítjuk egy másik memóriacímre.

- *double free*  
A másik gyakran előforduló hiba, ha kétszer akarjuk ugyanazt a memóriaterületet felszabadítani. Háromszoros vagy többszörös felszabadítás nem létezik, mivel már a másodikonál futási hibát kapunk.

### Feladatok:

- Foglaljunk le 12 byte dinamikus memóriát. Ellenőrizzük, hiba esetén írjunk ki hibaüzenetet. Írjunk rá három integer számot. Írjuk ki a memória tartalmát *printf*-fel. Szabadítsuk fel a pointeren keresztül.
  - Próbáljuk ki, mi történik, ha kétszer szabadítjuk fel a memóriát.
  - Próbáljuk ki, mi történik, ha nem 12 byte-t, hanem 12 Gbyte-t próbálunk foglalni.
  - Próbáljuk ki, mi történik, ha az első foglalás után egy másodikat hajtunk végre és az első foglaláshoz használt *pointer*t használjuk a másodikhoz is.
  - Definiáljunk egy *char\** *pointer*t, ami a foglalt memóriaterületre mutat. Írjuk felül a tárolt adatot az *abc* betűvel.
- Az önhivatkozó struktúrákat felhasználva, készítsük most el a teljes láncolt listát, azaz dinamikus memória kezelésével oldjuk meg a láncolt lista felépítését. A *node* adattagja legyen egyszerűen *int* típusú. Készítsük el az alábbi függvényeket:
  - *push\_back*: a láncolt lista végére fűz egy elemet, a bejövő *int* argumentummal feltöltve
  - *push\_front*: a láncolt lista elejére fűz egy elemet, a bejövő *int* argumentummal feltöltve
  - *pop\_front*: a láncolt lista első elemét törli és visszatér a *node*-ban tárolt adattag értékével
  - *pop\_back*: a láncolt lista első elemét törli és visszatér a *node*-ban tárolt adattag értékével
  - *insert*: az argumentumként kapott *node* (*pointer*) után helyez egy új *node*-t, a szintén argumentumként kapott adattal feltöltve
  - *erase*: az argumentumként kapott *node*-t (*pointer*) törliFigyeljünk a *node*-k egymásra hivatkozásaira. Minden műveletnél újra kell értelmeznünk a kapcsolatokat!

## File

A dinamikus memóriához hasonlóan végighaladunk a lépéseken. Azonban itt nem közvetlenül dolgozunk a tárolt adattal, azt különböző, a standard könyvtárakban megtalálható függvényekkel tudjuk írni és olvasni. Más nem. Ahhoz be kell töltenünk a memóriába (onnan regiszterekbe), majd visszaírunk azt a hard drive-ra. Az írást, olvasást egy *FILE\** típusú *pointer*en keresztül valósítjuk meg. Ez sok információt tartalmaz, többek között indikátorokat, a típusát, a file-n belüli pozíció mutatóját. Bővebben: <https://en.cppreference.com/w/cpp/io/c/FILE>.

A szükséges standard könyvtári header itt a *stdio.h*.

### Hozzáférés kérése:

*FILE\* fopen( const char\* filename, const char\* mode )*

A file nevének megadása egy karakterláncon keresztül történik, melynek létezésének ellenőrzése nem a függvény feladata. A *mode* argumentum a *printf* formátálási argumentumához hasonló, de

jóval korlátozottabb listából választhatunk: “r”, “w”, “a” az olvasás, írás és hozzáfűzés megadására, valamint mindegyik után tehetünk “+” karaktert, ekkor írni és olvasni is lehet. Megnyitható a file bináris formában is a “b” karakterrel. Bővebben: <https://en.cppreference.com/w/cpp/io/c/fopen>.

A függvény sikeres nyitáskor visszatér a file-ra mutató változóval. Ellenkező esetben null pointerrel. Például:

```
FILE const * const fp = fopen("temp.txt", "w");
```

### **Ellenőrzés:**

Nincs új a nap alatt, a null pointer ellenőrzést használjuk erre az esetre is: *if( fp == 0)*.

### **Felszabadítás:**

A nem-null file pointeren keresztül a file bezárása:

```
int fclose( FILE * fp )
```

A függvény siker esetén 0-val tér vissza, EOF-fal ellenkező esetben (ritkán ellenőrizzük). Bővebben: <https://en.cppreference.com/w/cpp/io/c/fclose>