6. gyakorlat

A gyakorlat témaköre a pointerek, tömbök és a memóriakezelés.

Pointerek

Idézzük fel a változók 4 alaptulajdonságát!

- név
- típus
- érték
- memóriacím

A változók értékei ezidáig számok, karakterek voltak. Lehet azonban mást is tárolni bennük: memóriacímet. A memóriacímet tároló változókat pointernek, mutatónak hívjuk.

A fiók analógián keresztül bemutatva (lásd 1. gyakorlat): a legtöbb fiókban számok és egyéb adatok találhatók, a pointer fiókokban viszont egy másik fiók sorszáma van.

A pointerek is változók, tehát ezeknek is van

- nevük
- típusuk: Van int pointer, double pointer, stb. Bármilyen adattípusú pointert létrehozhatunk. Az int pointer egész szám típusú memóriacellára mutat, a double típusú pointer double típusúra.
- értékük: A mutatott cella memóriacíme
- saját memóriacímük

Pointer készítése a gyakorlatban:

Változó létrehozásnál a `*` jelenti a pointert. Ez a csillag tekinthető a típus részének.

```
int *p;
```

Egy integer adatra mutató pointer készült, egyelőre viszont nem mutat sehova (értéke NULL).

Egyszerre több pointer létrehozásánál kicsit furcsa a szintaxis, mert a nevek elé kell csillagokat rakni:

```
int *p1, *p2, *p3;
```

Értékadás

Pointereknek kézzel nem tudunk értéket adni, mert nem ismerjük a program által használható memóriacímeket, és amúgy sem akarunk ezzel foglalkozni. A memóriakezelést az operációs rendszerre bízzuk.

A pointerek értékét másik változó memóriacímével, vagy memóriafoglalással (malloc, calloc) állítjuk be.

Változók memóriacímének megszerzése

A változók memóriacímét az `&` operátor segítségével tudhatjuk meg. Az operátor egy pointert készít, az adatának megfelelő adattípussal.

```
int szam = 420;
int *p = &szam;
```

A `szam` memóriacíme a `p` pointerbe került. Ki is írhatjuk.

```
printf("&szam = %p", p);
```

```
&szam = 0x7ffd1037bdbc
```

Többször lefuttatva változó értékeket láthatunk, mert az oprendszer máshol ad a programunknak memóriát.

Memóriafoglalás

Foglalhatunk új memóriát, ahova a pointerünk fog mutatni. Erre a malloc és calloc függvények szolgálnak.

> Megjegyzés: *Ez a fajta dinamikus memóriafoglalás veszélyes művelet, csak bizonyos célokra szokás használni. A konzi végén visszatérünk rá.*

malloc

Memóriafoglalás. Valahány bájtnyi memóriát kérhetünk, ezt `void *` típussal kapjuk meg, amit rögtön át kell alakítanunk megfelelő típusra. A malloc nem nullázza ki az átadott memóriaterületet, lehet benne szemét.

```
void *malloc(size_t size)
```

```
int *p;
p = (int *)malloc(4);
```

A `malloc(4)` 4 bájt memóriát foglal (ebben fér el egy int), az `(int *)` pedig átalakítja a kapott pointert int-re mutató pointer típusra.

Nem szeretjük kézzel kiszámolni az adattípusok méretét, ezért a 4 helyére a `sizeof(int)` kifejezést írjuk.

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

calloc

Nullázott memóriafoglalás. A calloc nem bájtonként foglal, hanem a megadott méretéből foglal n darab cellát. Ez is `void *` pointert ad vissza, át kell konvertálni. Az átadott memóriát kinullázza (minden bájtját nullával írja felül).

```
void *calloc(size_t nitems, size_t size)
```

```
int *p;
p = (int *)calloc(1, sizeof(int));
```

free

A malloc és calloc által foglalt memóriát a használat után fel kell szabadítani. Ezt a rá mutató pointer felszabadításával tehetjük meg.

```
free(p);
```

Felszabadítás nélkül a memória teleszemetelődik a lefoglalt celláinkkal, egy idő után a futtatás lelassul, majd ellehetetlenül.

Mutatott érték elérése (dereferencing)

```
int szam = 420;
int *p = &szam;
```

A p pointer mutat egy memóriacellára, amiben egy szám van. Ezt a számot a p poinerrel is el tudjuk érni. A mutatott adat eléréséhez a pointer neve elé egy csillagot (*) teszünk. Ezt a csillagot ne keverjük össze a pointer deklarációnál használt csillaggal.

> *A deklarációnál a csillag mondja meg a fordítónak, hogy egy pointert kell létrehoznia.*
>
> *A mutatott érték elérésénél egy létező pointer előtti csillag azt jelzi, hogy nem a pointerben lévő memóriacímet kérjük, hanem a címen lévő értéket.*

```
printf("A memóriacím: %p\n", p);
printf("A mutatott érték: %i\n", *p);
```

```
A memóriacím: 0x7ffd1037bdbc
A mutatott érték: 420
```

Pointer aritmetika

A pointereken lehet végezni pár műveletet.

Összeadás

Pointerekhez egész számokat hozzáadhatunk, ezzel ennyit lépünk a memóriában. A lépés nagysága az adattípusnak megfelelő, azaz `int *` pointernél 4 bájt egy lépés.

```
int *p = (int *)malloc(2 * sizeof(int));
*p = 1;
*(p+1) = 2;
```

A p mutat az első számra, a `(p+1)` a másodikra.

A p++ művelet eggyel előrébb lépteti a pointert, lényegében ez is egy összeadás.

Kivonás

Az összeadáshoz hasonlóan kivonni is lehet egész számot.

Különbség

Két pointer különbségét képezhetjük, ennek néha van jelentősége. A művelet eredménye a két cella közti adatok száma.

```
int *p_eleje = (int *)malloc(10 * sizeof(int));
int *p_vege = p_eleje + 9;

int meret = p_vege - p_eleje;
```

A `p_eleje` a lefoglalt 10 cella közül az elsőre mutat, a `p_vege` pedig az utolsóra. A két pointer különbsége megadja, hogy 10 cellánk van.

> Vigyázat! *A különbség eredménye nem bájtban van (egy int 4 bájt, a méret 40 bájt lenne). A pointer aritmetika az adattípus méretével számol.*

Pointerek összeadása értelmetlen, ezért nincs is ilyen művelet.

Tömbök

Egy darab adat változóban tárolását ismerjük. A legtöbb programban azonban nem csak 1-1 értéket akarunk tárolni, hanem sok, azonos értéket: egy mérési sorozatot, időpontok listáját, egy kép pixelértékeit. A sok adat együttes tárolásának legprimitívebb módja a tömb.

A tömb n darab **azonos típusú és azonos jelentésű** adatot tárol memóriafolytonosan: A memóriában az egyik után következik a másik, egy nagy egybefüggő *tömbként* jelennek meg.

A tömb a memóriában így jelenik meg:

```
----+-------+-------+-------+-------+-------+-------+----
... | adat0 | adat1 | adat2 | adat3 | adat4 | adat5 | ...
----+-------+-------+-------+-------+-------+-------+----
    ^
    kezdőcím
```

A tömb változó tulajdonképpen **pointer**. Kizárólag a tömb kezdőcímét tárolja. A tömb hosszát nekünk kell észben tartani.

Tömb készítése

Kétféle tömb van: statikus és dinamikus tömb. Ezek létrehozásuk módjában térnek el, egyébként a használatuk egyforma.

statikus tömb

A statikus tömböt akkor használjuk, ha fordítási időben ismerjük (vagy meg tudjuk tippelni) a tömb szükséges hosszát. Például: 100 számot akarunk letárolni, vagy egy email címet, vagy egy nevet. A memória nem drága, egy névnek foglalhatunk 60 karaktert is, akkor is ha általában csak 15-20 karaktert fogunk használni.

Létrehozása:

```
int tomb[100];
char nev[60];
```

A változó neve után szögletes zárójelbe írjuk, hogy hány darab hosszú tömböt szeretnénk létrehozni.

dinamikus tömb

A dinamikus tömböt akkor használjuk, ha csak futásidőben ismerhetjük meg a szükséges méretet: például a tárolandó adat méretét fájlból olvassuk be, vagy a felahsználó adja meg. Ezeket a fordítás idejében nem tudhatjuk.

A dinamikus tömböt malloc / calloc segítségével hozzuk létre, pointer típussal kell tárolnunk.

```
int *dinamikus_tomb;
size_t meret;
// méret megszerzése fájlból, felhasználótól ...
dinamikus_tomb = (int *)malloc(meret * sizeof(int));
```

A dinamikus tömböt a malloc miatt használat után fel kell szabadítanunk.

```
free(dinamikus_tomb);
```

Elemek elérése

A tömb elemeit indexeléssel érjük el. Az indexelés operátor a `[]`. Az indexelés a legtöbb programnyelvhez hasonlóan **nullától indul!**

```
int elem0 = tomb[0];
int elem1 = tomb[1];
int elem2 = tomb[2];

int dinamikus_elem = dinamikus_tomb[0];
```

Használatra a statikus és dinamikus tömbök egyformák.

Gyakori hiba, hogy nem létező elemeket akarunk elérni, ezt az operációs rendszerünk futásidejű hibával jutalmazza: Index Out Of Range.

> Megjegyzés: *Kapcsolat a pointerekkel. A `tomb[i]` művelet teljesen megegyezik a `*(tomb+i)` művelettel: mindkettő a `tomb` által mutatott kezdőcímtől `i` lépésre lévő cella tartalmát adja meg.*

Tömbök feltöltése

A statikus tömböknek lehet a létrehozásukkor értékeket megadni.

```
int szamok[4] = {2, 4, 6, 8};
```

Az elemszámot ilyenkor nem muszáj megadni, a fordító kiszámolja helyettünk.

```
int szamok[] = {2, 4, 6, 8, 10};
```

Nagyobb statikus tömböket, és dinamikus tömböket ciklusokkal szoktunk feltölteni. Jellemzően `for` ciklust használunk az alábbi példához hasonlóan.

```
int tomb[100];
for (int i=0; i<100; i++)
{
      tomb[i] = ... // érték megadása, pl. exp(i)
}
```

Az `i<elemszám` feltétellel a ciklus pont a tömb utolsó eleméig megy el.

Memóriakezelés

Ez a fejezet csak az érdeklődőknek szól, nem a tananyag része, viszont a megértést segíti.

A programunk a változóit a memóriában tárolja. A memórián két lényegében különböző területet használhat, ezek a stack és a heap.

stack

A stack, magyarul veremmemória egy egybefüggő blokk, amit a program induláskor megkap az oprendszertől. Ez egy nagy tömb a memóriában, melynek végére mutat egy **TOS** (Top of Stack) pointer. Képzeletünkben így körvonalazódhat a dolog:

```
      |  ....  |
      +--------+
      |        |
      +--------+
      |        |
      +--------+
      |        |
      +--------+
ToS ->|        |
      +--------+
```

Egy változó létrehozásakor az a ToS által mutatott memóriára tárolódik, a ToS pedig eggyel emelkedik.

```
int i = 420
```

```
      |  ....  |
      +--------+
      |        |
      +--------+
      |        |
      +--------+
ToS ->|        |
      +--------+
      |    420 |
      +--------+
```

A memóriafoglaláshoz nem kell várni az oprendszerre, hisz ez az egész memóriaterület a miénk. A foglalás gyors. A szokásos módon létrehozott változóink mind a stackre kerülnek.

Változók élettartama

A stacken létrehozott változók a scope végéig élnek. A scope végét a abból a függvényből való kilépés jelenti, amelyben a változót deklaráltuk. A függvényeken belül deklarált (lokális) változók a függvényből való kilépéskor törlődnek.

A megvalósítás úgy néz ki, hogy a függvénybe belépéskor a ToS mutató értéke eltárolódik. A függvény az eredeti ToS pozíció fölé tud lokális változókat létrehozni. A függvény kilépéskor visszaállítja a ToS pointert az eredeti pozícióba, így az efölötti értékek felszabadulnak, felül lehet írni őket.

heap

A heap memóriát az oprendszer kezeli, tőle kérhetünk itt szabad memóriát. A heapen létrehozott változókhoz az oprendszer keres egy megfelelő, szabad memóriablokkot. Ez egyrészt időigényes folyamat, másrészt az ész nélkül foglalt memória egyedi hibákhoz vezethet (memory leak, fregmentáció).

A heapen a malloc és a calloc segítségével foglalhatunk memóriát.

```
int *tomb = (int *)malloc(256*sizeof(int));
```

Az ilyen foglalások után mindig ellenőrizni kell, hogy valóban kaptunk-e memóriát.

```
if (tomb == NULL)
{
    //baj van
}
```

Változók élettartama

A heapen létrehozott változók a felszabadításukig élnek, ezt nekünk kell elintézni a free() függvénnyel. A programból való kilépéskor az oprendszer is törli a fel nem szabadított memóriát, de ne erre hagyatkozzunk!

Mikor kell a heapen foglalni?

- Dinamikus tömbök esetén. Ilyenkor fordítási időben nem ismerhetjük a tömb méretét, ezért csak a heapen foglalhatunk.
- Nagyon nagy tömbök esetén (több MB)
- Ha a scope-on kívül életben akarjuk tartani a változót, és máshogy nem tudjuk megoldani. (Erre ugye a másik mód a globális változó).

A heapen csak indokolt esetben foglaljunk, és mindig figyeljünk a felaszabadításra is!

Feladatok

1. Pointer gyakorlás

1. Hozz létre egy `val` nevű változót tetszőleges értékkel!
2. Készíts egy pointert, ami az előbb létrehozott adatra mutat!
3. Írd ki a pointerben tárolt memóriacímet (`"%p"`), és a mutatott adatot.
4. Változtasd meg a `val` értékét.
5. Ismételd meg a 3. pontban bemutatott kiírást. Ugyanazt az adatot éri el?

2. Tömb gyakorlás

1. Készíts egy 24 hosszúságú, karaktereket tároló (statikus) tömböt!
2. Töltsd fel az elemeit for ciklussal úgy, hogy a tömb az ABC-t tartalmazza!
   Használd fel, hogy `'A' + 1 = 'B'`, `'A' + 2 = 'C'`, *stb.!*
3. Most töltsd föl fordítva!

3. Reverse string

```
char szoveg[] = "Bolton";
char forditva[6+1];  // +1 záró karakter
```

1. Írj egy ciklust, ami a szöveget visszafelé beírja a `forditva` változóba!
2. Csináljuk meg ugyanezt pointerekkel!

```
char* forras = szoveg;
char* cel = forditva + 5;
```

Az `forras` a szöveg első karakterére, a `cel` a forditva utolsó karakterére mutat.

A pointerek léptethetők előre a `pointer++`, hátra a `pointer--` utasításokkal. Írj egy olyan ciklust ami ezek segítségével másolja át visszafelé a szöveget a forditvába.