Mutatók és memória

A C és C++ programozási nyelvek leghatékonyabb eszköze a közvetlen memória elérését lehetővé tevő mutatók és mutatóváltozók. (A jegyzetben mutatóként és pointerként is szerepelnek, természetesen ugyanazt jelenti mindkettő.) Más nyelvekben is fellelhetőek, azonban sok veszélyforrást is rejt a használatuk - elsősorban a figyelmetlen használatuk miatt -, így a magasabb szintű nyelvekben többnyire a programozók számára korlátozott megoldásokkal tudjuk csak használni vagy egyáltalán nem.

Minden program a memóriában van futás közben, minden változója, függvénye, adata (neve, string literálok, stb). A memóriának pedig minden byte-ja rendelkezik címmel, azaz a memóriában elfoglalt hellyel. Egy pointer, bővebben pointerváltozó nem más, mint ezt a memóriacímet jelentő **változó**. Ezt fontos realizálni, mivel onnan kezdve, hogy létrehozunk egy pointert, úgy az is a memóriában lesz tárolva egy adott memóriacímen – ami szintén lekérhető.

Mutatókhoz tartozó (memóriakezelő) operátorok:

- * operátor: címfeloldás, a mutatóváltozó által mutatott memória olvasható és írható közvetlenül
- & operátor: a memória címének lekérése, bármi is legyen az adott memóriaterületen

A címlekérő operátorral találkoztunk már a *scanf* függvény használatakor.

Feladatok:

- Írjuk ki egy tetszőleges változó címét. Ehhez definiáljunk egy *int a = 42* változót, majd printf(“%lu\n”, &a); utasítással kiíratjuk az *a* változó memóriacímét!

  Megjegyzés: A “%lu” helyett a gcc “%p”-t javasol warning formájában. A számítógépek architektúrája lehet 32 bites és 64 bites, ez pontosan a memóriacímek tartományát jelenti, és a pointerek méretét is meghatározza, előbbi esetben 4 byte, utóbbinál pedig 8 byte egy pointerváltozó mérete.

A szokásos, nem túl sok gyakorlati hasznot mutató és rém unalmas bevezetése a mutatóknak a lokális változók címének lekérése és tárolása egy mutatóváltozóban.

Mutatóváltozó deklarációja:

*típus *név*;

Mutatóváltozó definíciója, értékadása:

*típus *név = &változó*;
név = &változó;

Megjegyzések:

- A mutatóváltozó típusa a mutatott változó típusával kell megegyezzen! Ellenkező esetben a fordító warning-t ad. Nem hibát, ugyanis implicit típuskonverzió (*casting*) történik, de ez a casting általában hibát, súlyosabb hibát is eredményez.
- A deklarációnál lévő * nem keverendő össze a címfeloldó operátorral. Bár deklarációra és címfeloldásra ugyanazt a karaktert használjuk, meg kell különböztetni a két funkciót. A két funkció nem keveredik, így ütközni se tud a * két különböző értelmezése.
- Ahogy eddig megszoktuk C-ben, tetszőleges mennyiségű white space-t tehetünk az utasítások közé, sőt, ki is hagyhatjuk. Tehát az alábbi kifejezések tökéletesen megegyeznek:
  - int* p = &a;
  - int *p = & a;
  - int * p = & a;

Feladat:

- Egészítsük ki az előző kódunkat egy *int *p = &a* hozzáadásával, majd egészítsük ki a *printf*-t a p kiírásával: *printf(“%lu, %lu\n”, &a, p);*

- Többszöri futtatás során látható, hogy mindig más memóriacímre kerül az *a* változó.

- Ki is írhatjuk a *p* mutatóváltozónk méretét *sizeof(p)*-vel. Változtassuk meg a változónk és így a mutatóváltozónk típusát *short*-ra, *float*-ra, *double*-re és nézzük meg, mekkora a mutatóváltozó mérete.

- Most írjuk ki a mutató által hivatkozott memórián található változót címfeloldással:
  *printf(“%d”, *p);*

Pointer default értéke:
Ahogy a változóknál már láttuk, amennyiben nem adunk értéket egy változónak, úgy fordítótól függően az vagy inicializálva van (többnyire 0-ra), vagy valamilyen memóriaszemetet tartalmaz. Ez a mutatóváltozónál sincs másképp. Azonban itt különösen fontos, hogy inicializáljuk, ugyanis ha:

- 0-ra van inicializálva, azaz nullpointer esetén ha fel szeretnénk oldani, úgy nemlétező memóriacímen lévő adatot olvasnánk, írnánk
- memóriaszemét esetén az esetek többségében hozzánk nem tartozó, sőt, más program által használt memóriaterületen dolgoznánk

Mindkét esetben segmentation fault-tal terminál az alkalmazásunk. Ez a leggyakoribb hiba pointerek használatakor.

Feladat:

- Fenti programban ne inicializáljuk, vagy inicializáljuk 0-val a pointerváltozónkat, írjuk ki a legutóbbi feladattal a mutatóval hivatkozott területen lévő adatot (mondjuk *int* esetére).
  *int * p;*
  *// int * p = 0;*
  *printf(“%d”, *p);*

Pointer pointerre:
Ha már van egy pointerváltozónk deklarálva, úgy természetesen az is megtalálható a memóriában. Ennek a címe szintén lekérdezhető, szintén az & címoperátorral.

Feladat:

- Legyen egy mutatóváltozónk, pl *int *p = &a* az előző feladatból. Írjuk ki a *p* címét a printf-fel: *printf(“%lu”, &p);*

Ha le tudjuk kérni a mutatóváltozónk címét, úgy azt el is tudjuk tárolni egy változóban: ez egy mutatóváltozó lesz egy mutató(változó)ra.
Ennek **deklarációja, definíciója és értékadása**:

*típus ** név*;
*típus ** név = &(másik mutató)*;
név = &(másik mutató);

Továbbra is érvényes, hogy a mutató és a mutató a mutatóra típusának meg kell egyeznie, különben implicit casting történik, a fordító pedig erre figyelmeztet is.

Az adat kinyerése könnyedén alkalmazva az eddigi ismereteket:

1. A “pointer a pointerre” feloldható a * operátorral, ekkor megkapjuk a mutatott pointer értékét.
2. Az így feloldott pointer értéke ismételten feloldható a * operátorral, ekkor érjük el közvetlenül a változó memóriaterületét.

Feladat:

- Az előző kódunkat egészítsük ki egy újabb mutatóváltozóval, ami a már meglévő p pointerre hivatkozik és írjuk ki ennek az értékét :
  *int ** pp = &p;*
  printf(“%lu, %lu\n”, &p, pp);
- Írjuk ki a *pp* “pointer a pointerre” segítségével az *a* változónk értékét:
  *printf(“%d, %d, %d\n”, a, *p, **pp);*

A folyamat folytatható, tetszőleges *“pointer, ami pointerre mutat, … , ami pointerre mutat, ami egy memóriaterületre mutat”* készíthető. Gyakorlati haszna azonban egyre kevesebb, a háromszoros pointernek már csak elvétve van gyakorlati értelme.

Aritmetikai műveletek:
Mint unsigned long objektumok, korlátozott műveleteket tudunk végezni a pointerváltozókon:

- tudunk nemnegatív egész számot hozzáadni, illetve kivonni
- az egész számoknál megszokott módon, (possix és prefix) inkrementálni és dekrementálni

A számok körében értelmezett művelet a fordító által már le is vannak tiltva, mégha jogos is lenne a kérdés, ha adhatunk egy pointerhez számot, miért nem szorozhatjuk meg kettővel.

Feladat:

- Legyen két *int* változónk egymás után és egy pointerünk az első változóra mutatva. Írjuk ki a pointeren keresztül a hivatkozott memóriaterületet, majd írjuk ki az inkrementált pointer feloldásán keresztül az ott tárolt memóriát is. Pl.:

  ```
  int a = 42;
  int b = 137;
  int *p = &a;
  printf("%d\n", *p);
  ++p;
  printf("%d\n", *p);
  ```

  Megjegyzés*:* Nem feltétlenül tudjuk így kiírni így a második változó értékét, ez fordítás függő: nem garantált, hogy a két változó egymás mellé kerül a memóriában.

- Irassuk ki a pointerváltozó értékét az inkrementáció előtt és után is. Vegyük észre, hogy a két érték közötti különbség 4. Módosítsuk a változók és a pointer típusát *long*-ra és így irassuk ki a pointerváltozó értékeit: a különbség 8.

- Ugyan a szorzás művelete nem megengedett a pointerváltozókra, de körbejátszhatjuk. Amennyiben a pointerváltozó típusát *cast*-oljuk *unsigned long*-ra, úgy elvégezhetjük a szorzást (vagy bármilyen más egész típusra értelmezett műveletet), majd visszakonvertáljuk az eredeti pointer típusra. Például *float** típusra:

  ```
  float * p = &data;
  p = (float*) ((unsigned long)p * 2);
  ```

Logikai műveletek:
Ahogy két egész számot, úgy a memóriacímek értékeit is össze tudjuk hasonlítani a logikai műveletek segítségével. Például ha két pointer ugyanarra a memóriacímre hivatkozik, a szokásos == operátorral megvizsgálható. Ez fontos lesz a dinamikus memória kezelésében, ahol a memóriafoglalás után meg kell vizsgálnia a programnak, hogy az sikeres-e, azaz a visszatért pointer különbözik nullától. Példa:

```
double const * const p = (double*)malloc(10 * sizeof(double));
if ( p == 0)
        printf(“Memory allocation failed.”);
```

Feladat:

- Legyen két változónk, mindkettőre definiáljunk egy-egy pointert. Hasonlítsuk össze a két pointer értékét és írjuk ki, melyik változó van alacsonyabb memóriacímen. Példa:

```
int a = 1;
int b = 2;
int *pa = &a;
int *pb = &b;
if(pb > pa)
   printf("The 'a' variable is on lower memory address.\n");
else
   printf("The 'b' variable is on lower memory address.\n");
```

Típuskonverzió mutatókra:
Nem szabad összekeverni a kód írásánál a pointerváltozó típusának konverzióját és a mutatott memóriaterület típusának konverzióját!
A pointer típusa ugyanis *típus* *, vagy *típus* **, … vagy *típus* **..**, így a típuskonverzió explicit utasítása:

```
típus1 *p1;
típus2 * p2 = (típus2 *) p1;
```

A pointeren keresztüli változók típuskonverziója nem változik, de ekkor előbb fel kell oldani a pointerváltozót, tehát a * operátor a zárójelen kivül lesz! Fontos figyelnünk erre a könnyen elnézhető, de nagyon fontos részletre:

```
típus1 *p1;
típus2 var = (típus2) *p1;
```

A mutató mérete 64 bites rendszeren 8 byte, azaz maga a pointer is kasztolható *unsigned long* típusra. Amivel természetesen már tetszőleges, egész számokon értelmezett műveleteket tudunk végezni.

```
típus1 *p1;
unsigned long ulong = (unsigned long) p1;
```

Feladat:

- Az utolsó típuskonverzióval most már kiírhatjuk az inkrementálás, dekrementálás eredményét. Legyen egy változónk, aminek típusát változtatni szeretnénk (érdemes typedef segítségével flexibilissé tenni). Valamint két pointerváltozó, az első mutasson a változóra, a második pointer pedig az első pointer értéke plussz egy. Legyen egy unsigned long változónk, ami pedig tárolja a két pointerváltozó unsigned long konvertált különbségeit:

```
typedef int T;
T a = 42;
T * p1 = &a;
T * p2 = p1 + 1;
unsigned long diff = (unsigned long)p2 - (unsigned long)p1;
printf("%lu\n", diff);
```

Változtassuk meg a *T* típusaliast *char*, *float*, *double*, *long* típusokra, nézzük meg a különbségeket.

Megjegyzés: Amikor egy pointerváltozónak a típusát adjuk meg, nem csak a mutatott memóriaterület méretét és az ott lévő adat típusát adjuk meg, hanem a pointeren végzett művelet mértékét is. A memóriában haladás küönböző lehet eltérő típusok esetén, pl *char*-nál byte-nként, *int* esetében 4 byte-nként haladunk inkrementálás, dekrementálás esetén, vagy ha 2-t adunk a pointerváltozó értékéhez, úgy 2 byte-tal, illetve 8 byte-tal arrébb lévő memóriaterületre fog mutatni a pointerváltozó!

Tömbök és mutatók:

A tömbök nem mások, mint a memóriában **folytonosan** lefoglalt memóriablokkok. Például *char* típus esetén (a memóriacím mindig különböző):

0x7fff24064340	0x7fff24064341	0x7fff24064342	0x7fff24064343

Illetve int típus esetén ugyanazzal a memóriacímmel kezdve:

0x7fff24064340	0x7fff24064344	0x7fff24064348	0x7fff24064352

Egy mutatóval végig lehet haladni ezen a memóriaterületen, garantáltan a programunkhoz rendelt, a tömbhöz tartozó memórián fogunk végighaladni. A pointerváltozónkat természetes módon tudjuk inicializálni egy tömb első elemére:

```
type array[n] = { … };
type * p = array;
```

Ennél bonyolultabban is elkérhetjük a tömb első elemének címét az eddigiek alapján, ami ekvivalens a fenti definícióval:

```
type * p = & array[0];
```

Amennyiben nem az első tömbelemre szeretnénk a mutatónkat állítani, a hagyományos változó címének lekérésével, ahol az *idx* 0-nál nagyobb és a tömb méreténél kisebb:

```
type * p = & array[ idx ];
```

Ezt tudjuk egyszerűsíteni, a 0 tömbelemhez hasonlóan:

```
type * p = array + idx;
```

Feladat:

- Inicializáljunk egy 10 elemű tömböt tetszőleges értékekkel és típussal, majd irassuk ki a tömb elemeit egy pointer segítségével. Két módszer is lehetséges: vagy inkrementáljuk a

pointert lépésenként, vagy a pointerhez adjuk a ciklusváltozót és úgy oldjuk fel a memóriacímet.

- **Fun fact / megjegyzés.** A következő sorok ekvivalensek egy tömb pl. negyedik elemére:
  - array[4]
  - *(array + 4) // pointer és tömb ekvivalenciája
  - *(4 + array) // összeadás kommutativítása
  - 4[array] // memóriacím feloldása
    A tömbelem ilyen megadása a fordító által megengedett, de nem ajánlott ilyen kódot írni!

- Inicializáljunk egy 10 elemű tömböt tetszőleges értékekkel és típussal. Irassuk ki a tömb elemeit fordított sorrendben egy ciklussal, ahol a ciklusváltozó egy **pointerváltozó**! Példa:

```
short a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
for(short * p = a + 5; p >= a; --p)
   printf("%d, ", *p);
```

- **Haladó**: Visszatérünk kicsit a konverziókhoz. Léptessünk egy byte-t egy integer típusú mutatón! Hogy ellenőrízni tudjuk a léptetés mértékét, legyen egy kételemű *int* tömbünk, 257 és 0 értékekre inicializálva. A mutatónkat definiáljuk a tömb első elemének címére. Kövessük a következő aljasságot:

  ```
  p = (int*)((char*)p + 1);
  ```

  Irassuk ki a pointeren keresztül a memóriacímen található *int* értéket. Ellenőrízzük a bitenkénti alakját a 257-nek. *Megjegyzés*: A kételemű tömb értelme abban rejlik, hogy biztosan 0-ra van állítva minden bit a 0. tömbelem utáni memóriarészen, valamint hogy biztosan a programunkhoz tartozó memórián maradunk.

Felhasználói típusok és mutatók:

Nem csak built-in típusú változók memóriacímére hivatkozó mutatóink lehetnek, hanem user-defined típusú változókra is. A használatuk teljesen ugyanaz:

```
struct T
{
   type data;
};
struct T s = ...;
struct T * p  = &s;
```

A címfeloldás is hasonlóan működik és az adattag elérése is:

```
type a = (*p).data;
```

Ez a művelet elég gyakori, ezért saját operátort kapott: a nyíl (->) operátort. Ezzel egy pointeren keresztüli adattag elérése:

```
type a = p->data;
```

Feladatok:

- Készítsünk egy *struct* típust, amely tartalmaz egy *int* adattagot. Legyen egy 2 elemű tömbünk, típusa a most definiált *struct*. Cikluson keresztül adjunk a tömb minden elemének adattagjának értéket, mégpedig a ciklusváltozó értékét. Tipp a lehetséges elérési módszerekre (próbálj ki többet is):
  - array[i].data = i;
  - (pointer + i)->data = i;
  - pointer->data = i; p++; // ez akár a ciklusmagban is lehet
  - pointer++->data = i; // nehezen olvasható, érdemesebb külön írni

- Írjuk ki az így megadott tömb elemeinek adattagjait a *printf* segítségével.

  Megjegyzés: Amennyiben ugyanazt a pointert használjuk a két ciklusban, vegyük észre, hogy a pointer inkrementálása az első ciklusban maradandó (az utolsó két lehetséges elérési módszer esetében), azaz a tömb memóriaterületén túlra hivatkozó pointerünk lesz (potenciális segfault-tal)!

  Példa:

```
struct T a[2];
struct T *p = a;
for(int i = 0; i < 2; ++i)
    p++->data = i;
for(int i = 0; i < 2; ++i, ++p)
    printf("%d ", p->data);
```

- Zárójelezés elrontása! Írjunk egy programot, amiben kiírjuk az alábbi értékeket pointerek feloldásával, ahol i a ciklusváltozó: *printf(“%d”, *p + i)*, valamint *printf(“%d”, *(p + i))*. Figyeljük meg a különbségeket! Az első esetben előbb feloldjuk a pointert, azaz kiolvassuk a hivatkozott értéket, majd adjuk hozzá a ciklusváltozó értékét. A második esetben a pointerváltozóhoz adjuk a ciklusváltozó értékét, azaz léptetjük a pointert, majd feloldjuk azt és olvassuk ki a megfelelő memóriacímen található értéket.

Típusmódosítók mutatókok:

A négy típusmódosító közül a *const* módosítót nézzük meg részletesen, ez a legfontosabb mind közül és a legtöbb felhasználási lehetőséget adja. A pointerváltozó típusának deklarációjánál ugyanis nem csak a típus megadásához, hanem a pointer jelöléséhez is csatolhatunk konstans jelzőt. Az alábbi deklarációk különböző jelentéssel bírnak:

- *const type * p* vagy ezzel ekvivalensen *type const * p*
  ekkor a *p* mutatón keresztül az **adat** nem módosítható
- *type * const p*
  esetében a *p* **mutató értéke** nem módosítható

A két eset együtt is szerepelhet, azaz *type const * const p* deklarációnál se a pointerváltozó értéke nem módosítható és az általa hivatkozott adat se módosítható.

Feladat:

- Legyen két *int* változónk és egy pointerünk erre a változóra. Módosítsuk a változó értékét a pointeren keresztül, majd adjunk a pointerváltozónak másik értéket. Például:

```
int a = 0, b = 1;
int *p = &a;
*p = 42;
p = &b;
```

  - változtassuk meg a mutató deklarációját, az adatot védő *const* jelzővel

- változtassuk meg a mutató deklarációját, a pointer értékét védő *const* jelzővel

Nézzük meg, milyen üzenettel tér vissza a fordító mindkét esetben.

Mutató mint függvényargumentum

Eddig lokális változókon dolgoztunk, aminek felhasználási lehetőségei nem túl hasznosak, de legalábbis megkérdőjelezhetőek. A függvények argumentumaként való használatuk már jobban rá fog világítani, miért is olyan fontosak – főleg a hatékonyságot illetően.
Használatuk az eddigi függvényekről és a pointerekről tanultak alapján könnyen összetehetőek. Eddig is használtuk, lásd a *scanf* függvényt vagy a tömbök átadása egy függvény számára, ahol ugyan nem explicite, de valójában mutatót adunk át.
Az általános forma:

*return_type function (.., pointer_type * pointer, …);*

A függvényben pedig ugyanúgy használhatjuk, mint az előzőekben tárgyaltuk.

Ugyanígy általánosítható a többszörös pointerekre is a fenti szintaxis, ahogy a *const* típusmódosítóra is – sőt, kifejezetten célszerű is ez utóbbit használni, például elkerülve a Függvények tutorialban látott tömbök átadását és a függvényen belüli változtatását:

*return_type function (.., pointer_type const * pointer, …);*

A pointerváltozó konstansságát is célszerű megadni, de ha új hivakozást is állítunk át a pointerben, úgy az a hívó fél számára nem okoz változást.
Egy, a hívó fél oldalán lévő adat megváltoztatására egy függvényen belül úgy van lehetőségünk, hogy közvetlenül a hívó oldalán foglalt memóriát írjuk felül. Tehát egy “mezei” változót úgy tudunk változtatni, hogy annak a memóriacímét átadjuk a hívott függvénynek. Ez a szabály a pointerváltozóra is igaz: azaz egy pointer értékét a **hívó félnél** úgy tudjuk megváltoztatni, hogy a mutatót tároló memóriacímet adjuk át, azaz a függvény argumentuma egy pointer lesz a pointerre:

*return_type function (.., pointer_type ** pointer, …);*

Természetesen itt is van lehetőségünk *const* típusmódosító használatára, de már három helyen is. Egy mindenhol konstans függvényargumentum:

*return_type function (.., pointer_type const * const * const pointer, …);*

Általában persze azért adunk át dupla pointert, mivel a hívó oldalán lévő mutatót szeretnénk megváltoztatni a függvényen belül, tehát az utóbbi két *const* kulcsszót ilyenkor elhagyjuk.

Általánosítható a fenti dupla pointer többszörös pointer átadására is.

Feladat:

- Legyen a *main* függvényben két *int* változónk, tetszőleges értékre inicializálva.
  Készítsünk egy függvényt, ami felcseréli a két változó értékét (*swap*). Tipp: a függvénynek a két cserélendő változó memóriacímét kell átadni.
- Legyen a *main* függvényben két változónk, mindkettőre hivatkozzon egy-egy pointerváltozó. Készítsünk egy függvényt, ami ezt a két mutatót felcseréli, azaz ha a két változó *a* és *b*, a két mutató *pa* és *pb*, a függvény hívása előtt *pa* az *a* memóriacímét, *pb* a *b* memóriacímét tárolja, akkor a függvény hívása után a *pa* címfeloldása után a *b* értékét, a *pb*

pedig az *a* értékét adja vissza (*pointer swap*). Ellenőrizzük *printf*-fel a változók és a pointereken keresztül elérhető adatok értékeit a függvény hívása előtt és után is.

A mutatók használatának van még egy nagyon nagy előnye, mely nem is a beépített típusoknál válik nyilvánvalóvá, hanem a *struct*-ok, *union*-k és tömbök esetében. Ugyanis amikor egy függvényt meghívunk, akkor az átadott argumentumok lemásolódnak a függvényen belülre lokális változó formájában. Ez a beépített típusoknál 1, 2, 4, 8 byte-t jelent. Amikor viszont egy 100 byte méretű struct-t adunk át, úgy mind a 100 byte lemásolódik. Ha viszont erre csak egy pointert adunk át, akkor csak 8 byte másolása a költség. Persze ekkor fennáll a veszély, hogy a függvényen belül megváltozik az átadott struct valamilyen adattagja, de ez kiküszöbölhető a *const* kulcsszóval.

Tehát általános szabályként elmondható, hogy beépített típusoknál nem éri meg pointereket használni, de nagyobb felhasználói objektumok esetén igen. A tömböket a fordító automatikusan nem másolja, hanem az első elemre hivatkozó pointert másol át a függvénybe lokálisként. Természetesen ha a függvénynek a hívó fél oldalán lévő lokálist vagy még magasabb szinten lévő változót kell tudnia módosítani, úgy azt pointeren keresztül lesz képes.

A függvényekben, az argumentumként kapott mutatókat ugyanúgy kell kezelni, mint lokális mutatókat.

Figyelem! A függvényben lokálisan definiált, a stack-re hivatkozó mutatóval visszatérni segmentation fault-t okozhat futás közben. Általában kisebb és gyorsabban lefutó programoknál van esély sikeres futtatásra, amikor is a stack-n nem kerül olyan hamar használatban a függvény által használt memóriaterület.

Feladat:

- Legyen egy *int** típusú függvényünk, melyet a main-ben meghívunk, értéket adva egy *int** mutatónak. A függvény térjen vissza egy változó memóriacímével. Irassuk ki a mutatóértékét, valamint a hivatkozott értéket is. Például:

```
int * foo()
{
   int a = 42;
   return &a;
}
int main()
{
   int * p = foo();
   printf("%lu\n", p);
   printf("%d\n", *p);
   return 0;
}
```

- A gcc warning-t ad. Ezt ki tudjuk küszöbölni a függvényben definiált mutatóval, mely a változó memóriacímét tárolja és a függvény ezzel tér vissza:

```
int * foo()
{
   int a = 42;
   int * p = &a;
   return p;
}
```

- A program ezzel a “javítással” sikeresen lefordul és hiba nélkül is fut. Azonban nagyobb programoknál futási hibát tud okozni nagy eséllyel.

Önhivatkozó struktúrák

Mindkét összetett típusnál megoldható az önhivatkozás, azaz hogy tartalmazza önmagának egy példányát, vagyis egy arra mutató változót. Azonban a union esetében nem csak sok a futási hibát okozó probléma, de gyakorlati haszna is csekély. A struct esete viszont gyakran használt, lásd a bináris fa és a láncolt listák.
Hogy is néz ki egy ilyen típus definíciója:

```
struct Node {
  data_type data;
  struct Node* node1;
        ...
  struct Node* node1;
};
```

A … részen tetszőleges mennyiségű *Node** típusú, más ugyanilyen struktúrára mutató adattag lehet. Általában egy vagy kettő szokott lenni. A láncolt lista lehet egy-, illetve kétirányú: egy illetve két ilyen pointer adattagot tartalmazva. A bináris fa *node*-ja két pointer adattagot tartalmaz. A kétirányú láncolt lista és a bináris fa node-jai technikailag megegyeznek, csak más az adattagok elnevezése (láncolt lista: prev és next, bináris fa: left és right).

Feladatok:

- Legyen egy struktúránk, amiben egy önmaga típusát tartalmazó pointer adattagunk van, valamint egy *int* data adattag (egyirányú láncolt lista). A main-ben legyen egy 10 elemű tömbünk, amit ebből a típusból készítünk. Inicializáljuk egy ciklussal a *data* adattagokat véletlenszerű értékekkel (akár a random.h segítségével, akár beégetett konstansokkal).
- Készítsünk egy ciklust, ami az tömb elejéről végighaladva a megfelelő pointer adattagot beállítja a következő tömbelemre:

  ```
  node[i].next = &node[i + 1];
  ```

  - Alfeladat: írjuk át a szögletes zárójelek használata nélkül ezt a sort.

- Készítsünk egy függvényt, aminek átadjuk az első *node*-ra hivatkozó pointert. A függvény a láncolt listán végighaladva kiírja a megfelelő *node* data adattagját. A végighaladást a *next* adattagok segítségével hajtsuk végre. Például:

  ```
  while(node->next)
  {
      printf(“%d, ”, node->data);
      node = node->next;
  }
  ```

- Írjunk egy *sort* függvényt, ami rendezi a láncolt listát. A data adattagokat írjuk felül, ne a pointerváltozókat kötögessük át!

Az általános mutató

A *void** típust használjuk általános pointerként. Általános mutatók közötti műveletek esetén tudjuk használni, vagy amikor a hivatkozott adat típusa érdektelen, például memória bitenkénti másolása (*memcpy*). Habár általános, típusfüggetlen megoldásokban használható, mégse oldható meg vele a generikus programozás.

Feladat:

- Próbáljunk írni függvényt, mely két tetszőleges típusú változó felcserélését végzi el (mint a *swap* függvénynél, csak ez típusfüggetlenül).

Mutatótömbök:

Mutatókból készíthetünk tömböket is, mint bármely más típusból. Például:

*type * p [size];*

Feladat*:*

- Legyen három változónk. Készítsünk egy három elemű mutatótömböt, az elemeik pedig tartalmazzák a három változó címét.
  - Írjunk függvényt, ami argumentumként megkapja ezt a mutatótömböt (ez egy dupla pointer!), majd kiírja a pointereken keresztül a változók értékét.
  - Írjunk függvényt, ami a pointerváltozókon keresztül módosítja a változók értékét.

Haladó rész (nem kötelező): Függvénymutatók

Mint minden, a függvények is a memóriában foglalnak helyet, elsősorban a stack-n (kérdés: C++ nyelvben egy dinamikusan létrehozott osztálypéldány metódusai hova is kerülnek valójában?). És mivel egy pointer tetszőleges memóriacímre is tud hivatkozni, értelemszerűen következik, hogy egy függvényre is.

A hétköznapi programozói életben találkozunk is ilyenekkel, habár többnyire nem néven nevezve őket, hanem lambdáknak és callback függvényeknek nevezve őket. Egy ilyen függvénypointer ugyanúgy átadható egy másik függvénynek is, mint bármilyen pointerváltozó.

Deklaráció:

*ret_type (* function_pointer) (types_of_arguments);*

A lényeg az első zárójelekben van, ez jelzi a fordítónak, hogy nem sima mutatóváltozót szeretnénk, hanem függvénypointert.

Definíció / értékadás:

*ret_type (* function_pointer) (types_of_arguments) = function;*
*ret_type (* function_pointer) (types_of_arguments) = &function;*

és értékadás:

function_pointer = function;
function_pointer = &function;

A címlekérő operátor (&) elhagyható.

Használat:

A függvény meghívása a függvénypointeren keresztül ugyanúgy történik, mintha magát a függvényt hívnánk meg. Tehát ha a függvényt meghívjuk a következő módon:

function (arguments);

akkor a függvénypointeren keresztül:

function_pointer (arguments);

Példa:

- *int (*foo) (int)*: egy *int*-tel visszatérő és *int*-t váró függvényre mutató változó
- *void* (*bar) (int, double const * const)*: egy általános pointerrel visszatérő, *int*-t és *double const * const* argumentumokat váró függvényre mutató változó

Miért jó ez, ha tudjuk simán a függvényt is használni? Válasz: általános megoldások készítéséhez. Például láncolt listák *sort* függvénye, melyben a láncolt lista rendezését meghatározó feltételt mi adhatjuk meg egy tetszőleges függvénnyel. Ezt a sort függvény függvénypointer formájában veszi át és hívja meg!

Függvényeknél az argumentumlistába ugyanúgy megadni a függvénypointer argumentumot, mint a fenti függvénypointer deklarációnál. Azaz:

*ret_type function (… , fp_ret_type (*fp) (fp_arg_list), …);*

Feladat:

- Legyen egy elő-láncolt listánk, mint az önhivatkozó struktúra rész feladatainál. Írjunk egy függvényt, ami függvénypointert vár és az alapján rendezi be az elő-láncolt listát.
  Valahogy így néz ki:

  ```
  void sort (Node * const node, BOOL (*compare) (Node const * const, Node const * const))
  {
      …
  }
  ```

Todo feladat: függvény másolása a heap-re, majd meghívása onnan.