Típusok, adatok, utasítások

Az alapozások után végre elkezdhetünk C kódot írni. A C nyelv erősen típusos, azaz minden C-beli objektumnak van típusa. Objektum alatt itt most az adatokat (változók és konstansok) és a függvényeket tekintjük, utóbbi esetben a függvény típusa alatt a visszatérési értéknek a típusát értjük.

Rövidítés: A továbbiakban a fordítsuk le a programot és futtassuk a generált futtatható állomámyt a következő rövidítéssel értjük: ***C&R*** (compile and run).

Előfeltétel: Helloworld gyakorlat.

Előkészület: Készítsünk egy .c file-t, ami tartalmazza a *main* függvényt és az *stdio.h*-t include-ját.

Típusok

Nézzük meg milyen **típusok** lehetségesek:

- egész típusok
  - char: 1 byte
  - short: 2 byte
  - int: 4 byte
  - long: 8 byte
  - long long: 8 byte
- lebegőpontos típusok
  - float: 4 byte
  - double: 8 byte
- void
- összetett típusok:
  - array
  - struct
  - union

Megjegyzések:

- Az típusok méretei az architektúrától is függnek, az itt megadott értékek csak általában jellemzőek. A C nyelv biztosít egy beépített függvényt, mellyel le tudjuk kérni a foglalt memória méretét: ez a *sizeof(…)*.
- A *void* szigorúan véve nem típus, mivel nem lehet változónak vagy konstansnak ilyen típusa. A függvények előtt viszont lehet *void*, amikor nincs visszatérési értéke. Valamint, a pointerek részen látni fogjuk, hogy általános pointer típust tudunk majd jelölni *void ** módon.
- Az összetett típusokon belül a *struct* és *union* együtt a felhasználói típusok elnevezéssel is szoktuk illetni, míg a többi beépített típus.
- Bool típus nincs C nyelvben! Logikai értékek tárolására *int*-t használunk, vagy ha spórolni akarunk a memóriával, *char*-t.

Definíció és deklaráció

A változóknál és konstansoknál (nem a makróval), valamint a függvényeknél is meg kell különböztetnünk ezt a két utasításfélét, fogalmat.

Deklaráció: Amikor jelezzük a fordító felé, hogy ilyen változó, konstans (lásd lejjebb) vagy függvény létezik, használva lesz. Ha mégse, és -Wall kapcsolóval fordítunk, úgy figyelmeztetést

kapunk a változók, konstansok esetében (törölve az ilyen használaton kívüli kódokat, rövidebb gépi kódot kaphatunk, ami gyorsabb futást is tud eredményezni).

Definíció: Amikor értéket adunk a változónak, konstansnak, függvény esetében implementáljuk a függvény törzsét (kapcsos zárójelek közötti rész, utasítássorokkal feltöltve vagy üresen hagyva).

Feladatok:

- Nyissuk meg a .c kódunkat és csak deklaráljunk egy *int* változót, aminek értékét a *printf* függvénnyel kiíratjuk, *C&R*:

  ```
  int a;
  printf("a = %d\n", a);
  ```

  Megjegyzés:
  - A kiírt sor: *a = 0*. A legtöbb fordító a változók és konstansok értékét 0-nak definiálja, ha mi nem tesszük meg. Azaz, rejtett műveletként ilyenkor is definiáljuk. Ha nem volna definiálva, akkor valamilyen véletlen számot kapnánk futáskor, ami a változónak vagy konstansnak lefoglalt memóriaterületen maradt memóriaszemét.
  - Változók deklarálásakor érdemes egyben értéket is adni (azaz definiálni is).
  - Konstansok értékét kötelezően a deklaráláskor tudjuk csak megadni, utána módosítani nem is tudjuk (ezért konstansok, lást lejjebb a típusmódosítóknál).

- Nézzük meg a változónk méretét, valamint az int típus méretét a sizeof függvénnyel, *C&R*:

  ```
  printf("size of int = %d, size of a = %d\n", sizeof(int), sizeof(a));
  ```

  Megjegyzés:
  - Az *a* változó és az *int* típus mérete megegyezik (de ezen nem lepődünk meg).
  - A *sizeof* kimenete *long* típusú. Ezt a *%d* helyett *%ld*-vel jelöljük és tudjuk elkerülni a figyelmeztető üzenetet a fordítás során.
  - Játszunk a típusokkal, próbáljuk ki a *char*, *double*, stb típust is.

Típusmódosítók:

- *unsigned*: Nemnegatív értékek lesznek tárolva. Ez befolyásolja természetesen a tárolás mikéntjét is, mivel az előjeles esetben egy bitet az előjel értékére használunk el, ez többnyire az első bit balról. A maximális értéke a letárolható típusnak duplája az előjeles megfelelőjének (alsó határa 0).

- *const*: A változó értéke nem változtatható. Ilyenkor a definíció és deklaráció együtt kell lennie (C++ nyelven van lehetőség ilyenkor is változtatni).

- *static*: Két felhasználási lehetőségre szolgáló módosító. Ha globális változó előtt használjuk, akkor az nem érhető el más fordítási egységből, azaz .c fileból. A másik, gyakrabban használt eset, hogy egy függvény lokális változója előtt használva az egyszer inicializálódik, azaz ha a függvényt többször is meghívjuk, akkor az egyszer lesz deklarálva és definiálva. Nagyon jól használható számlálóként, például a függvény hívásaihoz. Másik remek alkalmazási terület, hogy globális változó helyett ilyen változót használunk, ezzel is kerülve a globális névtér szennyezését.

- *volatile:*
- *register*: A teljesség miatt került ide ez a két ritkán használt típusmódosító, a gyakorlat során mi nem fogjuk használni őket. A *volatile* kulcsszót akkor használjuk, ha más alkalmazással közös memóriaterületen tárolt változóról van szó. Ekkor a fordító nem optimalizálja ki. A *register* módosító pedig ajánlás a fordítónak, hogy a változót gyakran szeretnénk használni és a CPU egy regiszterébe tárolja le, ezzel gyorsítva a hozzáférést.

Feladatok:

- A .c kódban az *a* változónknak a típusát változtassuk meg *const*-ra, majd próbáljunk meg új értéket adni neki, *C&R*:

```
const int a = 2;
 a = 3;
```

- Töröljük az értékadást (*a = 3*), majd tegyük át a const módosítót az int kulcsszó mögé, C&R:

```
int const a = 2;
```

  Megjegyzés: A *const* módosító balra hat, azaz *const int == int const*, kivéve, ha a legelső kulcsszó a deklarációban. Ezt a későbbiekben jó tudni, amikor a pointerek is megjelennek: *int const * p* az *int*-re hat, nem a *-ra.

- Legyen két változónk, egyik *int* a másik *unsigned int* típusú, értékük mindkettőnek 0. Irassuk ki mindkét számot, az *unsigned int*-hez *%u*-t használjunk. Majd vonjunk le egyet mindkét számból és irassuk ki a ismét a két szám értékét:

```
int a = 0;
unsigned int b = 0;
printf("a = %d, b = %u\n", a, b);
a = a - 1;
b = b - 1;
printf("a = %d, b = %u\n", a, b);
```

  Megjegyzés:
  - Mindkét változó 4 byte-n tárolódik. Amennyiben túllépjük a megadott típusnak megfelelő maximális vagy minimális értéket, úgy túlcsordul (overflow) történik, a változó értéke a minimális vagy maximális értékre változik (az overflow jelzésére van egy kapcsoló a registerek egyikében, ott tudnánk assembly oldalról ellenőrízni, de C-ben egyszerű módon erre nincs lehetőségünk).
  - Valójában ha megcseréljük a *printf*-ben a *%u* és *%d* karaktereket, a fordító figyelmeztetés nélkül lefordítja sikeresen a kódot.
    De a későbbi algoritmusok miatt fontos észben tartanunk, hogy az *unsigned int* és *int* nem felcserélhető! (Lásd ciklus magjának indexelése.)
  - *unsigned int* helyett elegendő *unsigned* írnunk (az alapértelmezett *int* típus miatt a fordító tudja, hogy ekkor *int*-re értjük).

Összetett típusok

- **Tömbök**:
  Azonos típusú változók gyűjteménye, memórában egy összefüggő területet foglal el.

- típus változó[méret]; deklarálás, fordító többnyire feltölti 0-val
- típus változó[] = { érték1, … értékN}; fordító kitalálja a tömb méretét
- típus változó[méret] = { érték1, … értékN}; kevesebb érték esetén 0 a többi (ált.)

  Megjegyzés: A {} közötti értékadást inicializátorlistának hívjuk.

- egy adott elem elérése: tömb[index]

Feladatok:

- Legyen a kódunkban egy *char[3]* tömbünk:

  char a[3] = {'a', 'b', 'c', 'd'};

  Megjegyzés:
  - Fordításkor figyelmeztetést kapunk, ha több elemet adunk az inicializátorlistában mint a megadott tömb mérete.

- Legyen a .c kódunkban egy *short[5]* tömbünk, értékei {1, 2, 3, 4, 5}, irassuk ki *printf*-fel az első és harmadik elemet:

  short a[5] = {1, 2, 3, 4, 5};
  printf("a[0] = %d, a[2] = %d\n", a[0], a[2]);

  Megjegyzés:
  - Általában minden számolás 0-tól indul, a tömbök indexelése se kivétel.

- Irassuk ki a fenti tömb elemét az index = 5 esetre:

  printf("a[5] = %d\n", a[5]);

  Megjegyzés:
  - Mivel az indexelést 0-tól kezdjük, ezért az utolsó tömbindexünk a méret – 1 értékével egyezik meg, jelen esetben a 4.
  - Ennek ellenére a fordító nem jelez hibát, se figyelmeztetést nem ad, hogy a tömb méretét meghaladó tömbelemet kérünk el. A C-ben nincs méretellenőrzés, a programozótól feltételezi, tudja, mit csinál. (Persze mi magunk írhatunk ellenőrzést.)
  - A kiírt ilyen indexre általában még valamilyen meghatározott érték, nagy eséllyel a mi kódunk a memóriában elfoglalt eleme. Ha növeljük az indexet, eljutunk olyan pontra, ahol már véletlenszerű értékek kerülnek kiírásra, ez már a kódunk memóriában elfoglalt határán túl van, valószínűleg más alkalmazások használják vagy szabad terület memóriaszeméttel kitöltve.

- Irassuk ki a fenti tömb -1. indexű elemét:

  printf("a[-1] = %d\n", a[-1]);

  Megjegyzés:
  - Negatív indexek is lehetnek, továbbra sincs a fordítótól jelzés ennek hibájára. A kiírt értékekre szintén állnak a fenti, túl nagy indexes dolgok.

- Többdimenziós tömb:
  Használatuk ugyanolyan, mint az egyindexes tömbbé, memóriában szintén egy összefüggő területen van. Definiálásnál kell odafigyelni: egy tömb, ami egy tömb egy eleme, ami egy tömb eleme, stb.

  - Deklarálás: típus tömb[méret1][méret2]…[méretD];
  - Értékadás: tömb[méret1][méret2]…[méretD] = {{{}, {}, .. {}}, {}};
  - egy adott elem elérése: tömb[index1][index2]…[indexD]

  Feladat:
  - Legyen egy kétdimenziós *unsigned* tömbünk, mindkét dimenziójában hármas mérettel, feltöltve az alábbi értékekkel, majd változtassuk meg az második altömb harmadik indexű elemét:

    unsigned a[3][3] = {{11, 12, 13, 34}, {21, 22, 23}, {31, 32, 33}};
    *a[1][2] *= 2;*

    Megjegyzés:
    - Mindenben megegyeznek az egydimenziós tömbökkel, csak az indexelésük különbözik.
    - Egydimenziós tömbökként is felírhatjuk őket, kicsit bonyolultabb indexelési módszerrel.

      Például kétdimenziós tömbre:
      - deklaráció: tömb[méret1 * méret2]
      - tömbelem: tömb[index1 + index2 * méret1]

  Megjegyzés: három- és magasabb dimenziós tömbök nagyon ritkán vannak, a kétdimenziós tömböket mint mátrixokat szoktuk alkalmazni.

- **Struktúrák**
  Tetszőleges, már létező típus lehet benne, beleértve más struktúrákat, úniókat, de tömbök is szerepelhetnek.
  - *Szintaktikája*:

    ```
    struct StructNév
    {
       típus1 adattag1;
       típus2 adattag2;
         ...
       típusN adattagN;
    };
    ```

  - Ha már megvan a struktúránk, akkor egy változót deklarációja:

    struct StructNév structVáltozó;

Megjegyzés: Ha nem definiáljuk a változót, azaz adunk értéket az adattagjainak, akkor azt a legtöbb fordító a megfelelő típus 0 elemével tölti fel.

- *Definíció / értékadás:*

  structVáltozó = {érték1, érték2, … értékN};

  Megjegyzés: Ha nem definiáljuk a változót, azaz adunk értéket az adattagjainak, akkor azt a legtöbb fordító a megfelelő típus 0 elemével tölti fel. Tömbökkel megegyező mód.
- . *operátor* az adattagok eléréséhez, ha van egy definiált változónk:

  structVáltozó.adattagX

- **Uniók**

  Ritkán használt összetett típus, mi se fogjuk használni, csak a teljesség miatt van itt. Lényege, hogy az összes adattagjának egy, azaz egy memóriát foglal, aminek mérete megegyezik a legnagyobb méretű adattagéval.

  Szintaktikája hasonló a struktúrákéhoz:

```
union UnionNév
{
   típus1 adattag1;
   típus2 adattag2;
     ...
   típusN adattagN;
};
```

- **Typedef:** ezzel a kulcsszóval egy már létező típusra tudunk hivatkozni új névvel, a.k.a alias.

  A beépített típusoknál is egyértelműsíteni tudjuk, hogy egy adott esetben mit is akar jelenteni egy változó. De igazi haszna a struktúrák és úniók használatánál van, ugyanis segítségével elkerülhető a *struct* és *union* kulcsszavak felesleges használata.
  Szintaktika:

  typedef type aliastype;

  Pl.: *typedef struct S Struct;*

  Feladatok:

  - Készítsünk egy struktúrát és egy úniót, mindkettőnek legyen egy *char[15]*, egy *double* és egy *unsigned short* típusú adattagja:

```
struct S
{
   char c[15];
   double d;
   unsigned short s;
};

union U
{
```

```
    char c[15];
    double d;
    unsigned short s;
};
```

- Irassuk ki mindkét típusunk méretét a *sizeof* segítségével. *Megoldás: a programok gyorsasága miatt a memória blokkokra van osztva gyorsabb elérhetőség miatt, ezért a struct-k mérete felfele van kerekítve.*

Műveletek, utasítások

Matematikai műveletek

A megszokott aritmetikai műveleteket végezhetjük el: + - * / %. Utóbbi a modulo képzése. Egyszerűsítések az egy változón történő műveletek esetében:

- +=, -=, *=, /=
  a = a + 2;
  helyett írhatjuk, hogy
  a += 2;
- inkrementálás, dekrementálás: ++ és --, melyek eggyel növelik, illetve csökkentik az egész szám értékét
  - prefix:
    ++a, --a
    ekkor a változó értéke előbb inkementálódik, dekrementálódik, majd a beágyazó műveletek hajtódnak végre
  - possix:
    a++, a--
    ekkor a beágyazó műveletek hajtódnak végre, és azok után lesz csak a változó értéke módosítvainkrementálás, de

Precedencia:

A műveleteknek “fontossági sorrendje” van, igazodva részben a megszokott matematikában megszokottakhoz (pl szorzás előbb van mint összeadás). Zárójelekkel természetesen befolyásolhatjuk a műveletek elvégzésének sorrendjét. A további műveletekre is vonatkozik a műveletek precedencia szabálya.

https://en.cppreference.com/w/c/language/operator_precedence

- **Feladat**:
  - A .c file-ban definiáljunk egy *int a* változót 42 értékkel. Legyen még két *int* változónk, első legyen egyenlő 1 + *a*++-szal, a másik 1 + ++*a*-val. Irassuk ki minden lépés előtt a változókat (értelemszerűen, ha már definiáltuk őket).

```
int a = 42;
int b = a++;
printf("a = %u, b = %d\n", a, b);
int c = ++a;
printf("a = %u, b = %d, c = %d\n", a, b, c);
```

Megjegyzés: Nyomon követhető, hogy először a *b* értékét *a*-val tesszük egyenlővé, majd az *a* értékét növeljük possix inkrementálással. A *c* értékét pedig prefix inkrementálás után állítjuk az *a* értékére.

Bitműveletek

A változók értékei 0 és 1 formában vannak tárolva a memóriában, melyek között a legtermészetesebb műveletek a bitműveletek, integer típusokon hívhatjuk meg őket. A lebegőpontos számok tárolása elég kacifántos, de a fordító is hibaként értelmezi az ilyen utasítást.

Sokan elhanyagolják a bitműveletek jelentőségét, de valójában minden művelet bitműveletre van visszavezetve. Az egész számok összeadása például egy bitenkénti ÉS művelet. Mivel a processorok ezekkel dolgoznak, valójában ezek a leggyorsabb műveletek is.

- &, bitenkéni ÉS (AND)
- ^ , bitenkéni kizáró VAGY (XOR)
- | , bitenkéni VAGY (OR)
- ~, bitenkénti negálás (NOT)
- >> és <<, bit léptetés (shift)

- **Feladatok**:

  - Legyen a .c kódban két változónk, egyik *int* másik *unsigned*, értékük 0. Negáljuk bitenként az értéküket és írjuk ki őket.

    ```
    int a = 0;
    unsigned b = 0;
    a = ~a;
    b = ~b;
    printf("a = %d, b = %u\n", a, b);
    ```

    Megjegyzés:
    - Az *unsigned* változó negálásakor a maximális értéket kapjuk, ahogy vártuk is (a csupa 0-kból álló bitsor átmegy csupa 1-sekből álló bitsorba).
    - Az int változó esetében is a bitsorozat csupa 1-esből fog állni, de ennek értéke 1 lesz. Ennek oka, hogy a 2. komplementereként van értelmezve, az aritmetikai műveletek gyorsítása miatt (a -1 és 1 összeadása bit formában nézve: 11….11 + 00..01, melyen logikai ÉS-t alkalmazva 0-t kapunk egyszerűen, nem nézve az overflow-t).
      https://en.wikipedia.org/wiki/Two%27s_complement

  - Legyen egy változónk, értéke 1, melyet bitenként léptetünk, elsőként 1-t. Írjuk ki az értékét.

    ```
    int a = 1;
    a = a << 1;
    printf("a = %d\n", a);
    ```

    Változtassuk a léptetés mértékét.

    Megjegyzés:

- A 2 hatványait kapjuk eredményül, hiszen az 1 bitsorozata 00..001, melyet elléptetünk először egy helyiértékkel, azaz a kapott szám 00..010 lesz, majd még eggyel léptetve 00..100, és így tovább.
- Ha túl nagy értékkel léptetünk (bármely irányba) bitenként, akkor a végén 0-hoz jutunk.

Típuskonverzió

A beépített típusok között átjárhatóság van, ami természetesnek tűnik a számunkra, de ha belegondolunk, hogy általában máshogy vannak tárolva a memóriában, már nem is annyira egyértelmű. Egy 2 byte-s *short* és egy 4 byte-s *int* összeadásánál az egyszerűen megválaszolható kérdés, hogy mi legyen a felső 2 byte-n tárolt értékekkel, ha *int* típusú változóba szeretnénk tárolni. Fogósabb a kérdés, ha ugyanezt a két számot összeadva azt egy *short* típusban tárolnánk el. Itt adatvesztésről beszélünk, amit kontrollálnunk kell mint programozók.

A konverziók mikéntjéről a C nyelvben szerencsére nem kell nekünk gondoskodnunk, ezt a fordítók tudják. Ami a mi rendelkezésünkre áll, az a *kasztolás* (*cast*). Sőt, a legtöbbször – a beépített típusok esetében - nem is szükséges nekünk explicit leírni a kasztolás műveletét, azt a fordító implicit megteszi. Innen kezdve persze ez egy rejtett művelet, de sokszor egyszerűbben olvasható a kód, ha nem pakoljuk tele ezzel a típuskonverzióval.

Ha mégis szeretnénk mi irányítani a folyamat – ez sok esetben szükséges is, ha nem szeretnénk adatot veszíteni (pl *float* és *int* konvertálások során), a következőképpen tudjuk megtenni:

(újtípus)változó

Például:

```
int i = 42;
float f = (float)i;
```

Feladat:

- Legyen egy *int* változónk, értéke 4. Legyen egy float változónk is, melynek értéke az int változó osztva 10-zel. Irassuk ki printf-fel a float változót:

  ```
  int i = 3;
  float f = i / 10;
  printf("%f\n", f);
  ```

  Megjegyzés:

  - Természetesen ennek a kódnak is van haszna, de ha 0.3-t szeretnénk kapni, úgy explicit ki kell írnunk a típuskonverziót:

    ```
    float f = (float)i / 10;
    ```