Tartalomjegyzék

- 1 C tömbök
  - 1.1 Tömbök definiálása
  - 1.2 Tömb kapása függvényparaméterként
  - 1.3 Ellen?rzés és "http://wiki.math.bme.huundefined behaviour"http://wiki.math.bme.hu
  - 1.4 A fix méret kezelése
  - 1.5 C karakterláncok
- 2 További alapok
  - 2.1 Számok beolvasása
  - 2.2 Operátorok
  - 2.3 Könyvtárak

C tömbök

A C-ben a tömbök töltik be körülbelül ugyanazt a szerepet, mint python-ban a listák. Ez az alapvet? mód, amivel több dolgot egymás után rendezhetünk. Azonban a tömbök egy jelent?sen egyszer?bb konstrukció mint a listák. Ennek egyik része a típusosság következménye, egy tömbben mindennek egyforma típusúnak kell lennie, hisz el?re meg kell mondanom az elemek típusát. De vannak ezen túli különbségek is, amik még megnehezíthetik az életünket.

Tömbök definiálása

Alapvet?en kétféleképpen lehet tömböket definiálni a C-ben. Az egyik módszer az, ha a definíciónál egyb?l megmondod az értékeket is amik a tömbben vannak. Pl.:

```
int primek[] = {2, 3, 5, 7, 11, 13, 17, 19};
printf("%d\n", primek[0]);  // Ez a 2-t fogja kiirni
```

Ez a kapcsos zárójeles megadás csak ennél a definíciónál m?ködik, kés?bbi értékadásnál nem.

A másik módszer az, hogy a definíciónál csak a méretét mondom meg, pl.:

```
int primek[8];
primek[0] = 2;
// Ez nem mukodik: primek = {2, 3, 5, 7, 11, 13, 17, 19};
// Csak definicional
printf("%d\n", primek[0]);  // Ez a 2-t fogja kiirni
```

Ekkor az alapból benne lev? értékek nem definiáltak, véletlenszer?ek. Ne használjuk a benne lev? értékeket, amíg nem írtunk be arra a helyre már mi valamit.

Miután definiáltam a tömböt, a következ? dolgok igazak rá:

- A tömb mérete fix, nem változhat a futás közben. S?t, a hivatalos szabvány szerint nem is lehet oda a méret helyére változót írni, csak egy fix számot, esetleg egyszer? számítást mint "http://wiki.math.bme.hu4*10"http://wiki.math.bme.hu. (A gcc viszont elfogad egy változót oda.)
- Ezek a tömb típusok is mind különböz? típusok (általában úgy írják hogy pl. "http://wiki.math.bme.huint[]"http://wiki.math.bme.hu ha int tömbr?l van szó), tehát nem "http://wiki.math.bme.hukompatibilisek"http://wiki.math.bme.hu más típusokkal csak az explicit megengedett esetekben.

- A tömb tagjait a python-hoz hasonló szögletes zárójeles módszerrel lehet elérni, és itt is 0-tól (n-1)-ig vannak számozva. Itt nem m?ködnek az olyan negatív indexek, se az ilyen hosszabb részek kiválasztása kett?spontos kifejezésekkel. Csak egy darab egész szám lehet az index amit a szögletes zárójelbe írunk.
- A tömb hosszának meghatározására nincs egyszer? és mindig m?köd? módszer. Általában vagy egyértelm? (mert pl. mindig 3 dimenzió), vagy ha nem, akkor egy külön egész szám változóban tároljuk el, amit a tömbbel együtt odaadunk pl. függvényeknek meg ilyesmi.

Ez a szögletes zárójel egy módosító az alaptípushoz. Egy sorban több változót is lehet definiálni, és a hozzátartozó módosítók különböz?ek is lehetnek, így pl. ez:

```
int tomb_a[10], szam, tomb_b[10];
```

Az két tömböt definiál, és egy sima *int*-et. Ezt ugyan a nyelv megengedi, de általánosságban azt ajánlom hogy egy sorban csak egy változót definiáljunk, vagy legfeljebb azonos típusú és összetartozó dolgokat, de ilyen különböz? típusokat ne vegyítsünk.

Egy másik módosító a *mutató*, amit mi majd csak kés?bbi órán veszünk, és amit csillaggal jelölünk így:

```
int *mutato;
```

Csak azért említem most meg, mert sok szempontból ekvivalens a tömbbel, és ezért hibaüzenetben lehet hogy ezt látjuk amikor mi tömbökkel foglalkozunk, ekkor egyel?re mi ezt érthetjük úgy mintha az lenne ott hogy

```
int mutato[];
```

Tömb kapása függvényparaméterként

Ha függvényparaméterként szeretnénk tömböt kapni, akkor is használhatom mind a két változatot:

```
void f1(int v1[]) {
  printf("%d\n", v1[0]);
  // kod
}
```

vagy

```
void f2(int v2[3]) {
  printf("%d, %d, %d\n", v2[0], v2[1], v2[2]);
  // kod
}
```

Azonban a két módszer teljesen ekvivalens, a C fordító nem ellen?rzi hogy stimmel-e a tömb mérete, akkor se ha megadtuk. Az ott megadott méret tehát csak szándékot jelez, kvázi dokumentáció. Így ez a kód lefordul, csak aztán nem m?ködik jól:

```
void f2(int v2[3]) {
  printf("%d, %d, %d\n", v2[0], v2[1], v2[2]);
  // kod
}

void main() {
  int kettestomb[] = {30, 40};
  f2(kettestomb);
}
```

Ellen?rzés és "http://wiki.math.bme.huundefined behaviour"http://wiki.math.bme.hu

Fontos különbség a python és a C alapvet? filozófiái között, hogy mit csinál a program, ha hiba történik. Nézzük a következ?, egyszer?, C kódot:

```
int a[] = {10, 11};
int b[] = {20, 21};

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

Mit fog kiírni ez a kód? El?ször is, az biztos, hogy nem fog egy szép (vagy csúnya) hibaüzenetet dobni, hogy nem lehet -1-dik tagot kérni. Ahol én teszteltem, ott az alapbeállításokkal, ez azt írja ki, hogy 21, mert a memóriában éppen a *b* tömb adatai vannak az *a* mellett, ezért ha megyek egyet az *a* tömb 0. elemét?l visszafelé, ott a *b* tömb utolsó elemét találom. A C szabvány szerint ez "http://wiki.math.bme.huundefined behaviour"http://wiki.math.bme.hu. Ezzel a kifejezéssel még fogunk találkozni máskor is, úgyhogy kifejtem hosszabban mit jelent ez a C szabvány esetében.

A szabvány meghatározza hogy a helyes programoknak hogy kell viselkedniük, nem csak hogy mit kell kiírniuk, de a hatékonyságra és egyéb hasonló dolgokra is ad feltételeket. Azonban, ha a program nem helyes, ha a programozó hibát követ el, akkor sok helyen a szabvány azt mondja, hogy ebben az esetben a lefordított program "http://wiki.math.bme.huundefined behaviour"http://wiki.math.bme.hu-t tanusít. Ez elméletben a szabvány szerint azt jelenti, hogy ha ilyent csinálunk, akkor a szabvány felteszi a kezét, hogy akkor ? nem felel?s azért, hogy mi fog történni, akár az egész merevlemezünket is formázhatja a program.

Ezt persze nem úgy kell érteni, hogy azt leírni hogy "http://wiki.math.bme.hua[-1]"http://wiki.math.bme.hu veszélyes a gépünk egészségére. Azért nincs meghatározva ilyen esetben a viselkedés, hogy a különböz? fordítók a lehet? legnagyobb hatékonyságot érhessék el abban az esetben, ha helyes a program. Általában a fordítók sok esetet, ami a szabvány szerint "http://wiki.math.bme.huundefined behaviour"http://wiki.math.bme.hu, jobban definiálnak a dokumentációjukban, hiszen ?k már tudják, hogy az ? fordítójuknál ott mi fog történni. Pl. lehet hogy a gcc dokumentációjában le van írva, hogy "http://wiki.math.bme.huha az egészeket tároló tömbb?l olyan index-el olvasunk ki adatot, ami nem értelmes, akkor is vissza kapunk egy számot, aminek az értéke bármi lehet"http://wiki.math.bme.hu. (Bár ebben az esetben ez nem a teljes igazság, de ilyesmi.)

Ezzel ellentétben ugye a python kiírja hogy IndexError ezmegaz. De mit nyerünk ezzel az "http://wiki.math.bme.huundefined behaviour"http://wiki.math.bme.hu-rel, fejfájásokon túl? Hatékonyságot! Ha a python-on leírom azt, hogy "http://wiki.math.bme.hul[2]"http://wiki.math.bme.hu, akkor a processzoron sok utasítás fog lefutni, ami el?ször leellen?rzi hogy a 2-nek van-e értelme az adott listánál, ha nincs, akkor a hibát jelzi, ha van, akkor kiolvassa a megfelel? értéket. A C-ben egyetlen utasítás lesz, ami a megfelel? memóriacímet kiolvassa. Manapság azok az egyszer? ellen?rzések amiket a python csinál, a legtöbb esetben belefférnek az id?be, úgyhogy a legtöbb alkalmazásnál megéri ezt a kis hatékonyságot nagyobb biztonságra cserélni, de általában azokra a célokra használják a C-t ahol minden kis hatékonyságnak örülnek.

A fix méret kezelése

Mivel a tömb mérete a szabvány szerint még változó sem lehet, csak teljesen fix érték, ezért bajban vagyunk ha nem tudjuk el?re hogy mekkora tömbre lesz szükségünk, mert pl. a felhasználó által megadott adatoktól függ. Erre több megoldás is van, a szép megoldás majd az lesz, amit kés?bb tanulunk, hogy a memóriát kezeljük a programunkban. Addig is egyszer?bb, és sokszor teljesen megfelel? megoldás, ha azt mondjuk, hogy az adott program/függvény maximum 1000 elem? tömb kezelésére alkalmas.

```
// Dokumentaljuk is le, hogy max. 1000-el mukodik
```

```
int fun(int n) {
  if(n > 1000) {
    return -1;  // Ezzel a hibat jelezzuk
  }
  int tomb[1000];
  // A tenyleges szamolasok...
  // Amik a tomb-nek csak az elso n elemet hasznaljak
}
```

Ez ugyan pazarlásnak t?nhet, és ha nagyon hatékony kell hogy legyen a programunk, akkor nem is jó ötlet, de általában jó. (És ha hatékonyak akarunk lenni, akkor még mindig át tudjuk írni kés?bb.) És végezzünk egy gyors számolást hogy mennyire pazarlunk ezzel (a C-nek egy el?nye hogy az ilyesmi pontosan kiszámolható):

A gépünkben manapság mondjuk legalább 1 GB (GigaByte) memória van. Egy *int* típusú egész szám pontosan 4 byte memóriát foglal C-ben. Tehát ez az ezer elem? tömb még mindig csak 4 KB, a rendelkezésre álló memória 250000-ed része. (Ezzel ellentétben a firefox/chrome könnyen többszáz MB memóriát el tudnak vinni.) Amíg az így fixen lefoglalt memória 1 MB alatt van, még messze nem kell aggódni a hatékonyság miatt, ha amúgy úgy biztosra mehetek hogy az elég lesz.

C karakterláncok

A C-ben több különböz? típus is van az egész számok tárolására, attól függ?en hogy mennyi memóriát foglalnak. Nagyjából ezek vannak:

- A **char** típus -128 és 127 közötti számokhoz, 1 byte-on
- A **short** típus -32,768 és 32,767 közötti számokhoz, 2 byte-on
- Az **int** típus -2,147,483,648 és 2,147,483,647 közötti számokhoz, 4 byte-on
- A **long** típus, ugyanaz mint az *int* (A szabványban nem így van meghatározva, de jelenleg ez a helyzet.)
- A **long long** típus -9*10^18 és 9*10^18 közötti számokhoz, 8 byte-on (bár ez hivatalosan még a '89-es C szabványban nincs benne, csak a '99-esben, ismerik a fordítók)

Még vannak mindegyikb?l **unsigned** változatok is, ha negatív számokra nincs szükségünk, pl. az **unsigned int** típus 0 és 4,294,967,295 közötti számokat tud tárolni. Még egy fontos típus amit néha láthatunk a **size_t**, ez csak egy másik név valamelyik már említett típusra (általában az *unsigned int*-re), a beépített könyvtárak függvényei ezt használják a dolgok méretének a megadására. További leírást találhatunk pl. az angol wikipédián.

Na de ezek közül az egész típusok közül most a *char*-al foglalkozunk kicsit. A C-ben ha szöveget akarunk tárolni, akkor azt *char* típusu változók segítségével tesszük. Egy *char* változóban egy angol bet?t el lehet tárolni ASCII kódban kódolva. (Nem angol bet?k tárolását a python-ban sem tárgyaltam, de ott azért viszonylag kis szenvedéssel megoldható. C-ben elég nehéz, nem ajánlom.) Így a szöveget el lehet tárolni egy *char* tömbben, pl.:

```
char szoveg[] = "Szoveg!\n";
```

Ez egy 9 elemu *char* tomb lesz, és szoveg[0] az 83, mert 83 a nagy S-nek az ASCII kódja. Így nincsen a karakterlánchoz egy külön típus a C-ben, csak a *char* tömb típust, az egyéb használatai mellett, karakterláncok tárolására és kezelésére is használhatjuk. Tehát ha egy szöveget *char* tömbként akarunk nézni akkor arra használjuk a dupla idéz?jelet. Ha egy bet?t külön akarunk *char*-ként (és nem tömbként) nézni, ahhoz egyszeres idéz?jel kell. Pl.:

```
char betu = 'S';
// vagy
```

```
if(szoveg[0] == 'S') {
  printf("S-el kezdodik!\n");
}
```

Még egy dolgot csinál pluszban a dupla idéz?jel ami itt nem látszott, mégpedig azt, hogy a karakterlánc végére tesz egy 0-t. A 0-nak az ASCII táblában nincs jelentése (az van odaírva hogy "http://wiki.math.bme.hunull"http://wiki.math.bme.hu), hanem speciálisan arra szokták használni hogy jelezzék a végét a karakterláncnak. Így ha betünként akarunk feldolgozni egy karakterláncot, akkor addig kell menni amig egy 0 bet?t találunk. Pl.:

```
int feldolgoz(char[] szoveg) {
  int i = 0;
  while(szoveg[i] != 0) {
    // Csinalunk valamit szoveg[i]-vel
    ++i;
  }
}
```

Vagy, mivel a for ciklus rugalmas, azzal is ugyanezt megcsinálhatjuk:

```
int feldolgoz(char[] szoveg) {
  int i;
  for(i = 0; szoveg[i] != 0; ++i) {
    // Csinalunk valamit szoveg[i]-vel
  }
}
```

S?t, mivel a C-ben nincs külön igaz/hamis változó, hanem a 0 jelent hamisat, és minden ami nem 0 az igazat jelent, ezért még egyszer?bben így is írhatjuk:

```
int feldolgoz(char[] szoveg) {
  int i;
  for(i = 0; szoveg[i]; ++i) {
    // Csinalunk valamit szoveg[i]-vel
  }
}
```

Ezeket tudva érthetjük hogy a következ? két tömbmegadás miért jelenti pontosan ugyanazt:

```
char sz1[] = "abc\n";
char sz2[] = {97, 98, 99, 10, 0};
```

A karakterláncok kezeléséhez lesz néhány függvény a beépített könyvtárakban majd, egyel?re ennyit elég tudni róluk.

További alapok

Számok beolvasása

Ahogy a *printf* használható a parancssorra kiíráshoz, a scanf használható a parancssorról való beolvasáshoz. Egyszer? példa:

```
#include <stdio.h>

void main() {
  int n;
  printf("Irj be egy szamot!\n");
```

```
  scanf("%d", &n);
  printf("Azt irtad be, hogy %d.\n", n);
}
```

A *printf*-hez hasonlóan a "http://wiki.math.bme.hu%d"http://wiki.math.bme.hu az egész szám beolvasása, a "http://wiki.math.bme.hu%f"http://wiki.math.bme.hu a valós szám beolvasása. Amikor a változót ahova be akarunk olvasni megadjuk, elé kell tenni egy "http://wiki.math.bme.hués"http://wiki.math.bme.hu karaktert, egy "http://wiki.math.bme.hu&"http://wiki.math.bme.hu-t. Ha számokat akarunk beolvasni, akkor a scanf a whitespace-t figyelmen kívül hagyja, viszont ha más szöveget lát szám helyett, akkor nem megy. Így ahelyett hogy "http://wiki.math.bme.hu5"http://wiki.math.bme.hu beírhatom azt hogy "http://wiki.math.bme.hu 5"http://wiki.math.bme.hu, és akkor még m?ködni fog, de ha már azt írom be hogy "http://wiki.math.bme.hua5"http://wiki.math.bme.hu, akkor nem olvassa be sikeresen az 5-öt. Tehát parancssorról egy legfeljebb 100 elem? tömböt pl így olvashatunk be:

```
#include <stdio.h>

void main() {
  int n;
  int v[100];
  int i;
  scanf("%d", &n);
  for(i = 0; i < n; ++i) {
    scanf("%d", &v[i]);
  }

  // Feldolgozni a tombot
}
```

Így ez be tudja olvasni a következ? szöveget parancssorból:

```
3
10 11 12
```

Egy 3 elem? tömbbe aminek a 3 tagja 10, 11 és 12. (Igazából 100 a tömb mérete, de a kódban majd mindig csak az els? 3 elemével foglalkozunk, mindig csak n-ig megyünk, így effektíve 3 elem?nek vehet? a tömb.)

Ezt a példát azért mutatom külön, mert több labor feladatnál fog kelleni konkrétan.

Operátorok

Az operátorokat már eddig is használtuk, Ezek az ilyen jelölések mint a +, -, vagy a tömb indexelés. Az operátorok precedenciája mondja meg hogy amikor több operátor is szerepel egy sorban, akkor azt hogyan kell pontosan értelmezni. Itt a C-ben lev? összes operátor egy táblázatban:

Precedencia	Operátor	Rövid leírás	Asszociativitás	Példa
1	()	Függvényhívás	Bal	f(a)
	[]	Tömb-elérés		a[1]
	->	Mutatón keresztüli tag-elérés		ptr->b
	.	Objektumon keresztüli tag-elérés		a.b
	++	Posztfix növelés		a++
	--	Posztfix csökkentés		a--
2			Jobb	

	!	Logikai tagadás		!a
	~	Bitenkénti negálás		~a
	++	Prefix növelés		++a
	--	Prefix csökkentés		--a
	-	El?jel -		-a
	+	El?jel +		+a
	*	Dereferálás		*ptr
	&	Objektum címe		&a
	(típus)	Konverzió típusra		(int)a
	sizeof	Méret		sizeof(a)
3	*	Szorzás	Bal	5 * 3 == 15
	/	Osztás		5 / 3 == 1
	%	Maradékszámítás		5 % 3 == 2
4	+	Összeadás	Bal	5 + 3 == 8
	-	Kivonás		5 - 3 == 2
5	<<	Bitenkénti eltolás balra	Bal	5 << 1 == 10
	>>	Bitenkénti eltolás jobbra		5 >> 1 == 2
6	<	Kisebb	Bal	a < b
	<=	Kisebb-egyenl?		a <= b
	>	Nagyobb		a > b
	>=	Nagyobb-egyenl?		a >= b
7	==	Egyenl?	Bal	a == b
	!=	Nemegyenl?		a != b
8	&	Bitenkénti ÉS	Bal	6 & 5 == 4
9	^	Bitenkénti kizáró VAGY	Bal	6 ^ 5 == 3
10	|	Bitenkénti megenged? VAGY	Bal	6 | 5 == 7
11	&&	Logikai ÉS	Bal	(i < n) && good
12	||	Logikai(megenged?) VAGY	Bal	(i < n) || good
13	? :	if-then-else operátor	Jobb	(i < n) ? "http://wiki.math.bme.hukisebb"http://wiki.math.bme.hu "http://wiki.math.bme.hunem kisebb"http://wiki.math.bme.hu
14	=	Értékadás	Jobb	i += 5
	+=	Összeadás és értékadás		
	-=	Kivonás és értékadás		
	*=	Szorzás és értékadás		
	/=	Osztás és értékadás		
	%=	Maradékképzés és értékadás		
	&=	Bitenkénti ÉS és értékadás		
	^=	Bitenkénti kizáró VAGY és értékadás		
	|=	Bitenkénti		
	<<=			
	>>=			

		megenged? VAGY és értékadás Eltolás balra és értékadás Eltolás jobbra és értékadás		
15	,	Szekvencia operátor	Bal	a, b

Itt a magasabb precedencia azt jelenti, hogy azt az operátort hajtja el?bb végre ha lehet. Tehát pl. azt hogy:

```
a = 5 + 3 * 4;
```

úgy fogja a C értelmezni, hogy a szorzást hajtja el?ször végre, mintha az zárójelezve lenne így:

```
a = 5 + (3 * 4);
```

mivel a szorzásnak magasabb a precedenciája, 3, mint az összeadásnak, 4. Most még nem ismerünk a táblázatból minden operátort, de amire fel szeretném hívni a figyelmet, az a 14-es precedenciájúak, hogy minden aritmetikai operátor használható ilyen "http://wiki.math.bme.huváltoztatós"http://wiki.math.bme.hu formátumban, azok sokszor jól jönnek.

Könyvtárak

A C-ben is van néhány beépített könyvtár, bár jóval kevesebb dolgot tud, mint a python-hoz tartozó. A könyvtárakat (akár beépített, akár nem) a "http://wiki.math.bme.hu#include"http://wiki.math.bme.hu paranccsal tudjuk használni a programunkban. Miután az adott könyvtárat #include-oljuk, utána a benne lev? függvények elérhet?ek ugyanúgy mint a saját függvényeink. Ezért vigyázni is kell, ha egy könyvtárban van pl. "http://wiki.math.bme.huprintf"http://wiki.math.bme.hu nev? függvény, és a mi kódunkban is van "http://wiki.math.bme.huprintf"http://wiki.math.bme.hu nev? függvény, azt nem lehet máshogy megoldani, csak úgy hogy átnevezzük a saját függvényünket, nincs ilyen szép "http://wiki.math.bme.huas"http://wiki.math.bme.hu megoldás mint python-ban. Ezért érdemes a függvényeinke úgy elnevezni, hogy ne lehessen konfliktus. Pl. ha saját programot/könyvtárat írunk, abban minden függvény nevének az elejére tehetek egy jelzést, amit más valószín?leg nem használ: Ha eleve "http://wiki.math.bme.hucsirke_printf"http://wiki.math.bme.hu-nek neveztem el a saját függvényem, akkor nem valószín? hogy az bármilyen könyvtárral konfkliktusba kerül.

A C könyvtárai közül a következ?kb?l fogunk használni dolgokat a laborokon:

- <stdio.h> (Standard Input/Output): beolvasás, kiírás.
- <stdlib.h> (Standard Library): Alapvet? dolgok, mint számok karakterlánccá konvertálása és vissza, abszolút érték, ilyenek
- <math.h> (Mathematics): komolyabb matekos dolgok mint hatványozás, szinusz, koszinusz, logaritmus.
- <string.h> (Strings): Karakterláncok kezeléséhez kell? függvények, pl. azonosság ellen?rzése, hossz megmondása.

A C-nél nincsen központilag fenntartott hivatalos jó honlap ahol ezek megtalálhatóak, a hivatalos változat az a szabvány (a korábban említett 520 oldalas dokumentum). De több honlap is létezik, ami ugyan nem hivatalos, de jobban használható, ebb?l én a cppreference.com-ot ajánlom, az t?nik a legelfogadottabbnak jelenleg, nekem is az szokott a leghasználhatóbbnak t?nni, itt található a C-vel foglalkozó része.

Van itt egy 2 oldalas angol pdf C összefoglaló, amit nekem itt a BME-n adtak a Programozás tárgyamból, ami az eddig említett dolgokat elég tömören összefoglalja. (Mivel BME-s dolog, biztos nem bánják ha

továbbadom.) Ez pl. kinyomtatva laboron jó lehet a dolgok gyors megnézésére.