Mérnöki programozás 6

Szerkesztette: dr. Vass Péter Tamás

Mutatók

Mutató (pointer)

Egy változó definíciója tulajdonképpen a változó típusának megfelelő méretű memóriaterület lefoglalását eredményezi a program futása során.
A változó aktuális értékét ezen a területen tárolja a program kódolt formában.
Amikor egy változó azonosítóját használjuk egy értékadásban (akár annak baloldalán, akár jobboldalán), akkor tulajdonképpen a változóhoz tartozó memóriaterület tartalmát érhetjük el, ill. változtathatjuk meg.
A változó tehát mindig ugyanarra a memóriaterületre hivatkozik, tehát memóriafoglalás szempontjából egy statikusnak minősülő objektum.

Mutatók

Mutató (pointer)

Gyakran felmerülő probléma azonban, hogy előre nem ismerjük pontosan milyen mennyiségű adat számára kell, és mekkora területet lefoglalni, hanem csak a program futása során dől el (pl. adatok beolvasása fájlokból).
Ilyenkor dinamikusan kell memóriaterületet lefoglalni, ahová ideiglenesen tároljuk el az adatokat, majd ha már nincs szükség rájuk, a memória területet felszabadíthatjuk.
A memóriát úgy kell elképzelnünk, mint egy nagyon sok fiókos szekrényt, melynek minden fiókja 1 bájt kódolt információt képes tárolni.
A fiókok helyének egyértelmű azonosíthatósága érdekében minden egyes fiók rendelkezik egy címmel, ami egy címtartományból, kölcsönösen egyértelmű hozzárendelés alapján kerül kiosztásra.

Mutatók

Mutató (pointer)

A memóriaterület egyes bájtjainak eléréséhez tehát ismernünk kell a címüket.

A dinamikusan lefoglalásra kerülő memóriaterületen tárolt adatok elérésére nem alkalmazhatunk közönséges változókat, hiszen ezek számát, azonosítóját és típusát előre kell deklarálni.

Emiatt szükség van egy speciális változóra, amivel nem közvetlenül egy memóriaterületen tárolt adat értékére hivatkozhatunk, hanem egy memória címre, azaz a lefoglalt memórián belül egy meghatározott pontra.

A mutató (pointer) egy olyan változó tehát, amely memóriacímek tárolására alkalmas.

Mutatók

Mutató (pointer)

Egy mutató definíciójának általános formája:

```
típus *azonosító;
```

pl.

```
int *ptr;        /*egy int típusú változó címét tárolni
                 képes mutató*/
```

A * szimbólum jelzi a deklarációban, hogy nem egy közönséges változóról, hanem egy meghatározott típusú változó címének tárolására alkalmas mutatóról van szó.

A típus * (pl. int *) előírás tehát szétválaszthatatlan a deklarációban.

A definiált a mutató még nem "mutat" sehova, azaz nem tárol érvényes memóriacímet.

Mutatók

Mutató (pointer)

A mutatónak értéket kell adni ahhoz, hogy egy memóriacímre mutasson. Pl.

```
int a;
int *ptr;
a = 5;
ptr = &a;
```

Az & operátor adja meg a mutató számára az utána következő változó memóriacímét. (lsd. scanf függvény bemenő paramétereinél).

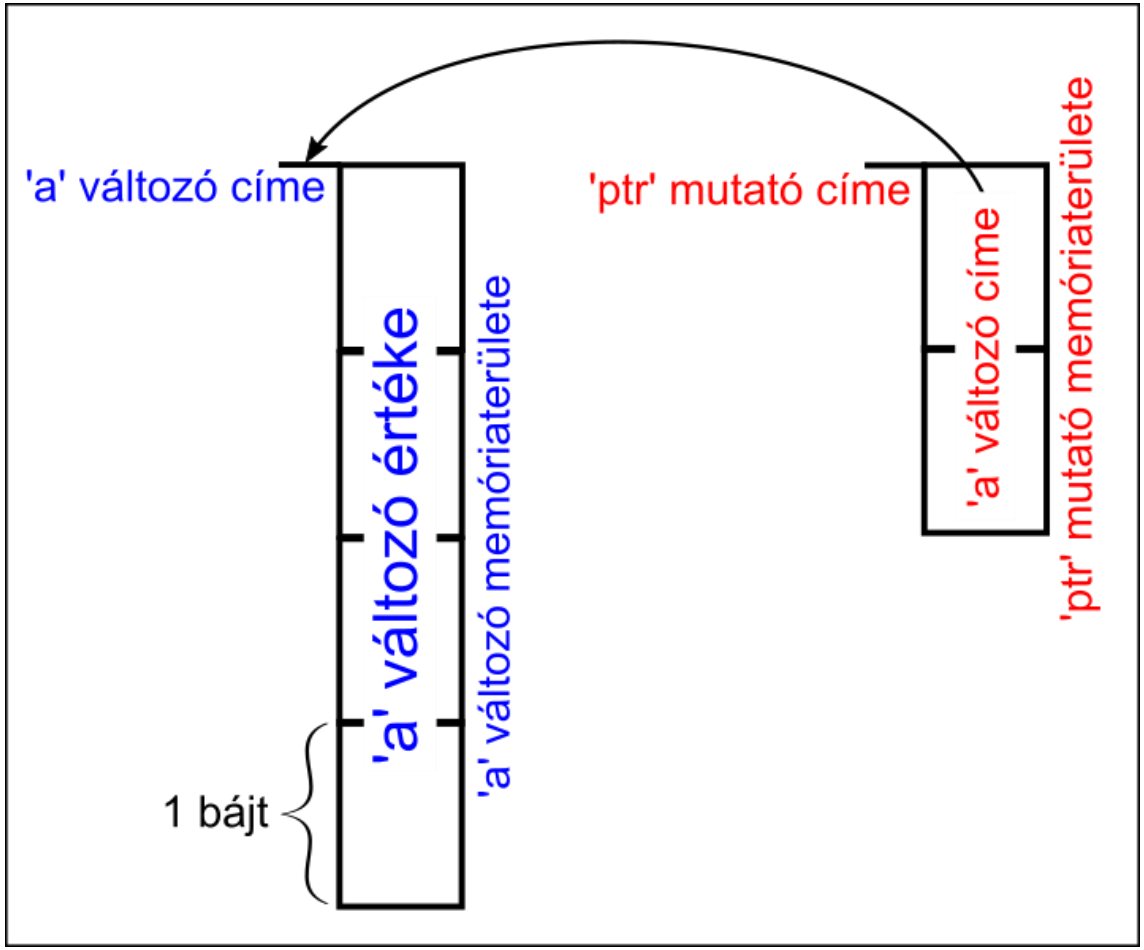
Ennek hatására a mutató értéke egy változóhoz tartozó (érvényes) cím lesz.

Ilyenkor azt mondjuk, hogy a mutató az adott címre mutat.

Mutatók

Mutató (pointer)

```
int a;
int *ptr;
a = 5;
ptr = &a;
```



Mutatók

Mutató (pointer)

Miután egy mutatóhoz hozzárendeltük egy változó memóriacímét, a mutató segítségével elérhetjük a változó értékét is. Pl.

```
#include <stdio.h>
main(){
        int a=5;
        int *ptr;
/*memóriacím hozzárendelés*/
        ptr = &a;
/*az a változó értékének közvetlen elérése a változó azonosítóval*/
        printf("a = %d\n", a);
/*az a változó értékének indirekt elérése a mutató segítségével*/
        printf("*ptr = %d\n", *ptr);
/*a mutatóban tárolt memóriacím (az a változó memóriacíme)*/
        printf("ptr = %p", ptr);
}
```

Mutatók

Mutató (pointer)

A * operátor neve indirekció operátor.

A mutató azonosítója előtt alkalmazva, a mutatóban tárolt memóriacímmel azonosított memóriaterületen tárolt adatot érhetjük el.

Alkalmazhatjuk kifejezésekben is, értékadó utasítás bal és jobboldalán egyaránt.

Segítségével megváltoztathatjuk a tárolt adat értékét is. Pl.

```
int a;
int *ptr;

a = 5;
ptr = &a;
a = *ptr + 2;
```

A mutató definíciójában alkalmazott * nem operátor, hanem a mutató típust jelző szimbólum.

Mutatók

Mutató (pointer)

Példa változó értékének megváltoztatására mutatón keresztül:

```
#include <stdio.h>
main(){
	int a=5;
	int *ptr;
	ptr = &a;
	printf("a = %d\n", a);
	printf("*ptr = %d\n", *ptr);
	*ptr = *ptr % 2;
	printf("a = %d\n", a);
	printf("*ptr = %d\n", *ptr);
}
```

Mutatók

Mutató (pointer)

Egy mutató segítségével több változót is elérhetünk a programban, ha a mutatóban tárolt memóriacím értékét megváltoztatjuk. Pl.

```
#include <stdio.h>
main(){
        double x=0.5, y;
        double *ptr;
        y = 3 * x;
        ptr = &x;
        printf("ptr = %p\n", ptr);
        *ptr = x + y;
        printf("x = %f\n", x);
        printf("y = %f\n", y);
        printf("*ptr = %f\n", *ptr);
        ptr = &y;
        printf("ptr = %p\n", ptr);
        *ptr = x + y;
        printf("x = %f\n", x);
        printf("y = %f\n", y);
        printf("*ptr = %f\n", *ptr);
}
```

Dinamikus memóriahasználat

Dinamikus memóriahasználat mutató segítségével

Dinamikus memóriahasználatra akkor van szükség, amikor előre nem tudjuk pontosan, hogy mennyi adattal kell műveleteket végeznünk, mert az adatok száma a program futása során válik ismertté.

Ilyenkor nem tudunk megfelelő számú változót definiálni előre a forráskódban.

A dinamikus memóriahasználat lényege, hogy a program futása során foglaljuk le a kívánt méretű memóriablokkot az adatok tárolásához.

A memóriablokk foglalása szabványos könyvtári függvénnyel történik (malloc), melynek deklarációját az stdlib.h fejlécfájl tartalmazza.

A legfoglalt memóriablokk kezdőcímét egy mutatóban tároljuk el.

Dinamikus memóriahasználat

Dinamikus memóriahasználat mutató segítségével

A dinamikus memóriafoglalás sikerességét mindig ellenőrizni kell a programban, mert ha nem sikerül, akkor a memóriablokkra vonatkozó további műveletek nem végezhetők el.

A memóriablokk kezdőcímét tároló mutató segítségével

adatokkal tölthetjük fel a memóriablokkot,

elérhetjük a tárolt adatokat,

és az értéküket meg is változtathatjuk.

Ha már nincs szükség a lefoglalt memóriablokkra, akkor azt fel kell szabadítanunk.

A következő példaprogramban a felhasználó döntheti el, hogy hány bemeneti adatot kíván megadni a program számára.

Ennek ismeretében a program már le tudja foglalni az adatok tárolásához szükséges méretű memóriablokkot.

A bevitt számadatok ezen a területen tárolódnak, majd a program kiszámítja, és meg is jeleníti a számok négyzeteit.

Végül a már szükségtelen memóriablokkot felszabadítja.

Dinamikus memóriahasználat

```c
#include <stdio.h>
#include <stdlib.h>

main(){
    int ndata, i;
    float *p;

    printf("Adja meg a beolvasni kivant adatok szamat! (max. 10) ");
    scanf("%d", &ndata);
    if (ndata<1 || ndata>10){
        printf("Hibas adatszam!");
        exit(-1);
    }
    p = (float *)malloc(ndata*sizeof(float));
    if (p==NULL) exit(-1);
    for (i=0; i<ndata; i++){
        printf("Adja meg az %d. adatot: ", i+1);
        scanf("%f", p+i);
    }
    printf("\nA beolvasott adatok negyzetei\n");
    for (i=0; i<ndata; i++)
        printf("Az %d. adat negyzete: %f\n", i+1, *(p+i)**(p+i));
    free(p);
}
```

Dinamikus memóriahasználat

Dinamikus memóriahasználat mutató segítségével

```
#include <stdlib.h>
```

A memóriablokk foglalására szolgáló malloc függvény miatt kell beépíteni a forráskódba a fejlécfájlt.

```
float *p;
```

A beolvasni kívánt adatok nem feltétlenül egész számok ezért float típusú adatok tárolására szolgáló memóriaterületek címeinek tárolására alkalmas mutatót kell definiálnunk.

Kezdetben a mutató egyetlen érvényes címet sem tartalmaz, azaz nem mutat semmilyen memóriacímre. Ilyenkor az értéke a NULL szimbolikus konstans értékével egyenlő. A NULL definícióját is az stdlib.h tartalmazza.

```
p = (float *)malloc(ndata*sizeof(float));
```

A malloc függvény hívása a memóriablokk lefoglalása érdekében. A malloc függvény bemeneti paraméterében meg kell adni a lefoglalni kívánt memóriablokk méretét bájtokban.

Dinamikus memóriahasználat

Dinamikus memóriahasználat mutató segítségével

A lefoglalni kívánt memóriablokk mérete az alábbi kifejezéssel számítható ki

```
ndata*sizeof(float)
```

amiben a `sizeof` operátor segítségével határozzuk meg egy float típusú adat tárolásához szükséges memóriaterületet bájtokban.

A beolvasni kívánt adatok számát a felhasználó adja meg, és az értéke az `ndata` változóban tárolódik. A beolvasásra kerülő adatok tárolásához szükséges memóriaterületet tehát *az adatszám * az egy adat tárolásához szükséges terület* összefüggés alapján kapjuk meg.

A malloc függvény miután lefoglalta a megadott méretű memóriablokkot, annak elejére mutató memóriacím értékével tér vissza a hívó függvényhez.

Mivel ezt a memóriacímet jelen esetben egy float típusú adatok memóriacímeinek tárolására alkalmas mutatónak kívánjuk átadni, előtte a malloc visszatérési értékét egy ún. explicit típuskonverziónak kell alávetni.

Dinamikus memóriahasználat

Dinamikus memóriahasználat mutató segítségével

Az explicit típuskonverzióval különböző típusú adatokat alakíthatunk át egymásba. Természetesen ez időnként információvesztéssel jár (pl. float típusú adat átalakítása int típusú adattá).

Az explicit típuskonverzió kerekzárójelpárba foglalva tartalmazza a megcélzott típus megadását (amivé át kell alakítani az adatot).

A zárójelpár megelőzi azt a változó vagy függvény azonosítót, amelynek értékét át kell alakítani.

Mivel a malloc függvény általánosan használható mindenféle típusú adatok tárolására szolgáló memóriablokkok lefoglalására, ezért a programban kell a visszatérési értékét olyan típusúra alakítani, amilyen típusú mutató fogadja a memóriablokk kezdőcímét.

Ezért kell alkalmazni a függvény azonosítója előtt a `(float *)` explicit típuskonverziót.

Ha a memóriafoglalás sikertelen akkor a mutató értéke NULL marad, azaz nem mutat sehova.

Dinamikus memóriahasználat

Dinamikus memóriahasználat mutató segítségével

Ezt használhatjuk fel a memóriafoglalás sikerességének ellenőrzésekor, az alábbi kódrészletben

```
if (p==NULL) exit(-1);
```

Ha nem sikerült lefoglalni az igényelt memóriaterületet, akkor a program leáll, és a main függvény -1 értékkel tér vissza az operációsrendszerhez.

Ha sikerült lefoglalni a kívánt méretű memóriablokkot, akkor megkezdődhet az adatok beolvastatása. Az első adat a memóriablokk kezdőcímével azonosított helyre kerül, melynek címét a p mutató tárolja. A második adat egy float típusú adat tárolásához szükséges memóriaterülettel távolabbi címmel kezdődő területre kerül, melyet a p+1 kifejezés értéke ad meg. Ha ugyanis egy mutatóhoz hozzáadunk egy egész számot, akkor az eredmény a mutatóban tárolt címtől távolabbi cím értéke lesz. A távolság mértékét bájtokban úgy kapjuk meg, hogy az egész számot megszorozzuk a mutatóhoz tartozó adattípus tárolásához szükséges terület bájtokban kifejezett hosszával.

Dinamikus memóriahasználat

Dinamikus memóriahasználat mutató segítségével

Megjegyzés: a memóriablokkon belül visszafelé is mozoghatunk a címekkel, ha a mutató aktuális értékéből egy egész számot kivonunk.

A beolvasott adatok megfelelő helyekre kerülését az alábbi módon biztosítjuk

```
scanf("%f", p+i);
```

az i értéke azért kezdődik 0-val a for ciklusban, hogy az első adat a p-ben tárolt címre kerüljön.

Figyeljük meg, hogy itt nem kell az & (címe) operátort használnunk a scanf függvény második bemeneti paraméterénél. Ugyanis most nem egy változókat, hanem a mutató segítségével előírt címeket adjuk meg közvetlenül.

Végül kiszámítjuk a beolvasott adatok négyzeteit és kiíratjuk az eredményeket egymás alá.

Az egyes számadatokat a mutató segítségével indirekt módon érjük el az indirekció (*) operátorát alkalmazva

Dinamikus memóriahasználat

Dinamikus memóriahasználat mutató segítségével

Az egyes számadatok négyzeteinek számításához alkalmazott kifejezés:

```
*(p+i)**(p+i)
```

A zárójelen belüli kifejezések segítségével adjuk meg a soron következő számadatot tároló memóriaterület kezdőcímét.

Ha a mutatóra, ill. a mutatót tartalmazó kifejezésre alkalmazzuk az indirekció operátorát (*), akkor magát a tárolt adatok érhetjük el.

A fenti kifejezésben természetesen a második * a szorzás műveletét jelenti. Az első és a harmadik * indirekt elérést (nem változó azonosítón keresztüli elérés) biztosít az adathoz a mutatóban tárolt címen keresztül.

Tömbök

Tömb (array)

A tömb egy olyan leszármaztatott adattípus, amely azonos típusú adatok meghatározott méretű halmazának tárolását és elérését biztosítja.

Legegyszerűbb változata az egydimenziós tömb, ami azonos típusú adatok sorának feleltethető meg.

Ha számokból áll az egydimenziós tömb, akkor a matematikában ismert vektor fogalom C nyelvű reprezentációjának tekinthető.

A tömböket, mint minden változót, használatuk előtt definiálni kell a forráskódban.

Egydimenzós tömb definíciójának általános formája:

típus tömb_azonosító[méret];

A típusban adjuk meg, hogy a tömb milyen típusú adatok tárolására szolgál. A void (üres) típuson kívül minden más típus alkalmazható.

A méret egy pozitív egész szám, vagy egy olyan kifejezés, melynek értéke egy pozitív egész számnak felel meg.

Tömbök

Tömb (array)

Példa:

```
int x[5], y[10];
double z[10];
```

A definíció révén tulajdonképpen a tömbhöz rendelünk egy memóriablokkot, melynek kezdőcímére fog mutatni a tömb azonosítója.
A tömb és egy értéket kapott mutató között tehát az a hasonlóság, hogy mindkettő rámutat egy címre, azaz tárol egy memóriacímet.
Míg azonban a mutatónak új értéket is adhatunk a programban - azaz más-más memóriacímekre mutathat rá - addig a tömb azonosítója mindig ugyanarra a kezdőcímre mutat.
A tömb definiálása tehát statikus memóriafoglalással jár együtt, ami azt jelenti, hogy a tömbhöz rendelt memóriablokk a program indulásától a végéig lefoglalva marad.

Tömbök

Tömb (array)

A statikusan lefoglalt memóriablokk mérete a tömbben tárolni kívánt adatok típusától és a tömb méretétől függ.

Például egy n elemű int típusú adatokat tároló tömbhöz tartozó memóriablokk mérete az

```
n * sizeof(int)
```

kifejezéssel számítható ki.

A tömb definíciójakor kezdőértékek is adhatók a tömb elemek számára. Például:

```
float x[4] = { -1.56, 0.87, 6.78, -23.87 };
char w[] = { 'c', 's', 'o', 'k', 'i' };
```

Több elemet nem szabad megadni az értékadó listában, mint a tömb mérete, de kevesebbet igen. Ilyenkor a fennmaradó tömb elemek értéke 0 vagy meghatározatlan érték lesz.

Ha kezdőértékeket adunk meg a tömb definíciójában, akkor nem kötelező a tömb méretét megadnunk. Ilyenkor a tömb mérete akkora lesz, ahány elem számára kezdőértéket adtunk meg.

Tömbök

Tömb (array)

Gyakori eset, hogy a tömbök elemei a programban végrehajtott számítások eredményeképpen kapnak értékeket.

Amikor értéket akarunk adni a tömb valamelyik elemének, akkor a tömb elemre az indexével hivatkozunk az értékadó utasítás bal oldalán. Például:

```
x[2] = a * b;
```

A fenti példában a 2-es indexű tömbelem valójában a tömb harmadik eleme, mert az elemek indexelése 0-tól kezdődik és n-1-ig tart, ahol n a tömb mérete. A szögletes zárójelpár ilyenkor az indexelés operátorának felel meg.

Egy tömb elemeinek számított értékekkel való feltöltésére leggyakrabban a for ciklust használjuk fel.

A következő példában látható, hogy nem csak értékadásban, hanem kifejezésekben is indexelésekkel hivatkozunk egy tömb elemeire.

Tömbök

```
#include <stdio.h>
#include <stdlib.h>   /*az exit függvény miatt kell*/
#include <math.h>     /*az sqrt függvény miatt kell*/
#define N 4    /*a tömb méretét itt egy szimbolikus konstanssal
                adjuk meg*/

main(){
    int i;
    float data[N];
    printf("Adjon meg %d nem negativ szamot!\n", N);
    for (i=0; i<N; i++){
        printf("A(z) %d. szam: ", i+1);
        scanf("%f",&data[i]);
        if (data[i]<0){
            printf("Hibas adat!\nA program leall.");
            exit(-1);
        }
    }
    printf("\nA megadott szamok negyzetgyokei.\n");
    for (i=0; i<N; i++)
        printf("A(z) %d. szam gyoke: %f\n", i+1, sqrt(data[i]));
}
```

Tömbök

Tömb (array)

A mutatókhoz való hasonlósága a tömböknek abban is megmutatkozik, hogy a tömb egyes elemeinek a memóriacímeit is elérhetjük a tömb azonosító segítségével.

Például:

ha az x –el azonosított tömbnek n eleme van, akkor az x+i kifejezéssel a tömb i-edik elemének kezdőcímét érhetjük el a tömb memóriablokkján belül. Az i index természetesen 0 és n-1 között változhat.

A mutatóknál alkalmazott indirekció operátorát (*) pedig az i-edik indexű elem értékének az elérésére is felhasználhatjuk a következő formában: *(x+i), ami megfelel az x[i] tömbelem hivatkozásnak.

A következő program az előzőnek egy másik változata, amiben nem az indexelés operátorát alkalmazzuk a tömb elemeinek elérésére, hanem az indirekció operátorát. További különbség, hogy a beolvasásnál alkalmazott scanf függvény második bemeneti paraméterénél nem kell az & (címe) operátor alkalmaznunk, mert közvetlenül meg tudjuk adni az i-edik indexű tömbelem kezdőcímét az x+i kifejezéssel.

Tömbök

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define N 4
main(){
     int i;
     float data[N];
     printf("Adjon meg %d nem negativ szamot!\n", N);
     for (i=0; i<N; i++){
          printf("A(z) %d. szam: ", i+1);
          scanf("%f",data+i);
          if (*(data+i)<0){
               printf("Hibas adat!\nA program leall.");
               exit(-1);
          }
     }
     printf("\nA megadott szamok negyzetgyokei.\n");
     for (i=0; i<N; i++)
          printf("A(z) %d. szam gyoke: %f\n", i+1,
                    sqrt(*(data+i)));
}
```

Tömbök

Tömb (array)

Az egydimenziós tömböket gyakran használják ún. sztringek (charakterek összetartozó sorozata) létrehozására, tárolására, és rájuk vonatkozó műveletek elvégzésére.

Amikor egy sztring számára char típusú elemeket tartalmazó tömb formájában helyet foglalunk le, akkor gondolnunk kell arra, hogy minden sztringet egy ún. null karakterrel (\0) kell lezárni.

Tehát mindig legalább egy karakterrel több adat számára kell memóriaterületet lefoglalni a tömb definíciójakor, mint amennyit maximálisan megengedünk a sztring számára.

Karaktertömböt definiálhatunk kezdőérték nélkül vagy azzal együtt. Pl.

```
char str1[51];
char str2[51] = {'A','z',' ','i','d','o',' ',
                 's','z','e','p','.','\0'};
char str3[ ] = {'A','z',' ','i','d','o',' ',
                 's','z','e','p','.','\0'};
char str4[51] = "Az ido szep.";
char str5[ ] = "Az ido szep.";
```

Tömbök

Néhány, sztingek beolvasására és kiírására szolgáló könyvtári függvény (az stdio.h fájlban vannak deklarálva):

`scanf`

`printf`

karaktertömbben tárolt sztring beolvasásához és kiíratásához alkalmazandó formátum specifikáció: %s

`char *gets(char *s)` beolvas egy sort a billentyűzetről az s karaktertömbbe, és visszatér az s tömb kezdőcímével. Sikertelen művelet esetén NULL a visszatérési érték.

`int puts(const char *s)` az s karaktertömb tartalmát kiírja a képernyőre és újsor karakterrel zárja le. Visszatérési értéke hiba esetén EOF, egyébként pedig egy nem negatív egész szám.

Tömbök

Néhány sztring-kezelő könyvtári függvény (a string.h fájlban vannak deklarálva):

`char *strcpy(s2, s1)` az s1 karaktertömb sztringjét átmásolja az s2 karaktertömbbe, a \0 karakterrel együtt. Az s2 sztring kezdőcímével tér vissza.

`char *strcat(s2, s1)` az s1 karaktertöm sztringjét az s2 tömb sztringjének végéhez fűzi, és az s2 tömbe kezdőcímével tér vissza.

`size_t strlen(s)` az s karaktertömbben tárolt sztring hosszával (karaktereinek számával) tér vissza.

`int strcmp(s2, s1)` összehasonlítja az s1 és az s2 karaktertömbökben tárolt sztringeket. Nulla értékkel tér vissza, ha két sztring azonos.

Tömbök

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

main(){
        char str[101];
        int i, n;

        printf("Gepeljen be egy szovegsort!\n");
        gets(str);
        n = strlen(str);
        printf("A begepelt szoveg hossza: %d\n", n);
        printf("A begepelt szoveg nagybetus valtozata:\n");
        for (i=0; i<n; i++)
                putchar(toupper(str[i]));
        putchar('\n');
}
```

Tömbök

Többdimenziós tömbök

Lehetőség van többdimenziós tömbök létrehozására és használatára is.
Többdimenziós tömb definíciójának általános alakja:

```
típus tömb_azonosító[méret1] [méret2] … [méretn];
```

Tehát minden dimenzió számára meg kell adni a mértet.
Példa egy int típusú adatokat tárolni képes kétdimenziós tömb definíciójára:

```
int tomb2D[3][4];
```

Kétdimenziós tömbök segítségével lehetővé válik a matematikában használt mátrix reprezentálása a programjainkban.
A fentebb definiált tömbnek 3 sora és négy oszlopa van, tehát egy 3x4-es méretű mátrixnak felel meg.
A számára lefoglalt memóriablokk bájtokban értelmezett mérete az alábbi kifejezéssel számítható ki:

```
3 * 4 * sizeof(int)
```

Tömbök

Többdimenziós tömbök

A kétdimenziós tömb definiálásakor már kezdőértékeket is adhatunk a tömb elemeinek. Például:

```
float matrix[3][4]= {{-1.2, 5.8, 0.6, 11.5},
                     {8.5, -5.7, 14.2, 0.1},
                     {3.7, -4.5, 5.1, -6.5}};
```

A sorokat nem kötelező kapcsos zárójelek közé zárni, de a tömb szerkezetének felismerését segíti a fenti definíciós formátum.
A kétdimenziós tömb elemeire két index értékének megadásával hivatkozhatunk. Az indexek értéktartománya 0-val kezdődik és a sorok, valamint az oszlopok számánál eggyel kisebb értékekkel záródik.
Tehát a `matrix[2][3]` elem értéke a fenti definíció alapján `-6.5`.
A kétdimenziós tömbök elemein végzett műveletekhez leggyakrabban két egymásba ágyazott for ciklust használunk (pl. elemek feltöltése adatokkal, számítások az elemekben tárolt adatokkal, és az elemek értékeinek megjelenítése).

Tömbök

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main(){
        int i, j, N=5, M=3, matrix[N][M];
        int a=1, b=100;

        srand(time(NULL));
        printf("Pszeudo-veletlen szamok generalasa "
                "%d es %d kozott\n", a, b);
        for (i=0; i<N; i++)
                for (j=0; j<M; j++)
                        matrix[i][j] = rand() % b + a;
        printf("A matrix elemeinek ertekei:\n");
        for (i=0; i<N; i++){
                for (j=0; j<M; j++){
                        printf("%d\t", matrix[i][j]);
                }
                putchar('\n');
        }
}
```

Tömbök

Többdimenziós tömbök

Az előző program véletlen egész számokkal tölt fel egy 5x3-as mátrixot, melynek elemeit rendezett módon írja ki a felhasználó számára.

A véletlen, pontosabban mondva, ál-véletlen számok létrehozására szolgáló könyvtári függvények a rand és az srand. Mindkettő az stdlib.h fájlban van deklarálva.

A rand függvény egy ál-véletlen egész számot hoz létre a 0 és RAND_MAX tartományban. A RAND_MAX szimbolikus konstans definíciója szintén az stdlib.h fájlban található, értéke általában 32767.

A függvény a generált szám értékével tér vissza a hívó függvényhez.

Deklarációja:

```
int rand(void)
```

Tömbök

Többdimenziós tömbök

Az srand függvény az ál-véletlen szám létrehozásához biztosít kezdő értéket.

Deklarációja:

```
void srand(unsigned int start)
```

Egyetlen pozitív egész számot igényel bemeneti paraméterként. Ennek értéke befolyásolja a rand() függvénnyel létrhozott véletlenszámok értékét.

Ha biztosítani akarjuk, hogy a rand függvény minden futtatáskor más véletlen szám sorozatot hozzon létre, akkor a program elején más-más értékkel kell meghívni az srand függvényt.

A programban ezt úgy biztosítjuk, hogy az srand függvényt az aktuális időpont másodpercben kifejezett értékével hívjuk meg.

Ezt a time függvény segítségével kapjuk meg, amely a time.h fájlban van deklarálva.

Tömbök

Többdimenziós tömbök

A programban az ál-véletlen számok tartományát megváltoztatjuk. Az alsó és felső határokat az *a* és *b* változók tartalmazzák.

Az alábbi kifejezéssel biztosítjuk, hogy a generált számok a kívánt tartományban legyenek:

```
rand() % b + a
```

A program kettős for ciklusokat használ a kétdimenziós tömb eleminek értékadásához és az értékek kiíratásához.

Az i ciklusváltozót a sorok indexeléséhez, a j-t pedig az oszlopok indexeléséhez használjuk fel.