Tartalomjegyzék

- 1 Preprocesszor
  - 1.1 include
  - 1.2 define
    - 1.2.1 igaz/hamis példa
- 2 Mutatók
  - 2.1 Egyszer? változókra mutatók
    - 2.1.1 A * és & operátor
    - 2.1.2 Példa a mutató használatára
    - 2.1.3 Mutató mint függvényparaméter
    - 2.1.4 Null mutató
  - 2.2 Mutatók és tömbök
  - 2.3 Változók élettartama

Preprocesszor

A C programok fordítása két részb?l áll. El?ször lefut az úgynevezett "http://wiki.math.bme.hupreprocesszor"http://wiki.math.bme.hu, és utána fut le a rendes fordító. A preprocesszor olyan utasításokat kap, amiket a programon mint szövegen hajt végre, úgy, hogy még nem próbálja értelmezni hogy C-ben mit jelent az. Mindjárt meglátjuk hogy ennek milyen következményei vannak.

Persze általában nem kell tudnotok hogy egy fordító hogy m?ködik belül, de mivel itt a preprocesszor eléggé mást és máshogy csinál mint a rendes fordítók, ezért külön tanuljuk azt ami hozzá tartozik.

A preprocesszornak a kett?skereszttel kezd?d? sorok szólnak. Mivel ezek nem C parancsok egészen pontosan, más nevük van, *preprocesszor direktíva*knak hívjuk ?ket. Emiatt van az is, hogy a C-re vonatkozó általános szabályokkal ellentétben ezeknél számít hogy egy parancs mindig egy sor, aminek az els? karaktere kett?skereszt, tehát a preprocesszor az újsorokat figyelembe veszi ellenben a C fordító másik felével, ahol az is csak egyfajta whitespace. És ezért nem kell az ilyen sorok végére pontosvessz?.

include

Az egyetlen példa amit láttunk eddig erre, az az "http://wiki.math.bme.hu#include"http://wiki.math.bme.hu parancs. Az #include az azt csinálja, hogy egy másik fájl teljes tartalmát beilleszti annak a sornak a helyére. tehát amikor azt írjuk, hogy

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

Akkor a fordító fog egy "http://wiki.math.bme.hustdio.h"http://wiki.math.bme.hu nev? fájlt (ami a fordítón belül van), és annak a teljes tartalmát beilleszti ennek a sornak a helyére. Ez a fájl az, ami a különböz? beépített függvények deklarációit tartalmazza, ezért használhatjuk ?ket, miután volt ilyen #include sor.

Ezzel lehet saját másik fájlokat is együtt felhasználni, akkor nem kacsacs?röket, hanem dupla idéz?jeleket kell használni a fájlnévnél, így:

```
#include "masik_fajl.h"
```

Az a különbség a kett? között, hogy az idéz?jellel megadott fájlokat el?ször az aktuális könyvtárban keresi, míg a kacsacs?rrel megadottakat el?ször a fordító könyvtárában keresi.

define

Vegyük ezt a példát az el?z? el?adásról:

```
// 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
}
```

Itt az a probléma, hogy az "http://wiki.math.bme.hu1000"http://wiki.math.bme.hu mint limit, kétszer is le van írva. Ha meg akarjuk növelni a limitet 2000-re, könnyen elfelejthetjük, hogy két helyen is módosítani kell, és csak az egyiket írjuk át. Azonban, a szabvány szerint, nem használhatunk változót a tömb méretének megadására, ezért az sem m?ködik, hogy eg változóba tároljuk el. Ehelyett a "http://wiki.math.bme.hu#define"http://wiki.math.bme.hu direktívát használhatjuk egy konstans meghatározására. Itt van ez jól megvalósítva:

```
#define TOMB_MAX_MERET 1000

// Dokumentaljuk is le, hogy max. 1000-el mukodik
int fun(int n) {
  if(n > TOMB_MAX_MERET) {
    return -1;  // Ezzel a hibat jelezzuk
  }
  int tomb[TOMB_MAX_MERET];
  // A tenyleges szamolasok...
  // Amik a tomb-nek csak az elso n elemet hasznaljak
}
```

Ezzel elértük hogy egy helyen elég módosítani a méretet.

Azonban ezzel a #define-nal vigyázni kell. Ezt a preprocesszor csinálja, ami nem érti a C kódot, és annak különböz? részeit, bárhol ahol azt a nevet látja amit megadtunk neki, oda behelyettesíti azt a másik bet?sort amit megadtunk neki. Így pl. ha ezt írjuk:

```
#define a xxx

int alap(int u) {
  int az = 0;
  // stb...
}
```

Akkor ebben a kódban a preprocesszor a kis "http://wiki.math.bme.hua"http://wiki.math.bme.hu bet? minden el?fordulását kicseréli, tehát "http://wiki.math.bme.huxxxlxxxp"http://wiki.math.bme.hu nev? függvényt definiáltunk, és azon belül egy "http://wiki.math.bme.huxxxz"http://wiki.math.bme.hu nev? változónk van. Ez még túlélhet? lenne, de ha pl. az "http://wiki.math.bme.hui"http://wiki.math.bme.hu bet?t #define-olom, akkor már azt se írhatom le sikeresen hogy "http://wiki.math.bme.huint"http://wiki.math.bme.hu, így egész változókat nem tudok megadni.

Tehát a lényeg az, hogy olyan nevet adjak a #define-al megadott konstansnak, ami nem fordulhat el? máshol a kódban. Az általánosan elfogadott szokás az, hogy a #define-al megadott konstansoknak a neve csupa nagybet?b?l áll, ezt ajánlom követni.

igaz/hamis példa

Mutatok egy példát ilyen preprocesszorral megadott konstansra, amit sok tényleges C programban is használnak. Ha zavar hogy az igaz/hamis értékek tárolására is egész számokat használunk, és a 0-val jelképezzük a hamist, és általában az 1-el az igazat, akkor ehelyett be lehet vezetni egy-egy konstanst ezekre a célokra, és ezzel lehet hogy átláthatóbbá tesszük a kódunkat:

```
#define TRUE 1
#define FALSE 0

int prime(int szam) {
  int i;
  for(i = 2; i < szam; ++i) {
    if(szam % i == 0) {
      return FALSE;
    }
  }

  return TRUE;
}
```

Így az olvasó számára látszik hogy az "http://wiki.math.bme.hu1"http://wiki.math.bme.hu az mikor az 1 mint szám, és mikor jelenti azt hogy "http://wiki.math.bme.huigaz"http://wiki.math.bme.hu. A C fordító teljesen ugyanazt a kódot látja, mint ha 1 és 0 lenne ott, tehát neki mindegy.

Mutatók

A C-ben az ilyen mutable és immutable dolgok helyett van egy speciális külön típus, a mutató. Már az el?z? el?adáson említettem a szintaxist, hogy hogy néz ki, a *-gal jelöljük. Tehát egy *int* típusú változóra mutató mutatót így definiálunk:

```
int *mutato;
```

Egyszer? változókra mutatók

A * és & operátor

A mutató típusú változók nem arra valók hogy adatokat tároljanak, hanem más változókra való hivatkozások, és a másik változó tárolja a tényleges adatokat. Így érhet? el olyasmi dolog amit a python-ban a mutable változókkal, hogy két különböz? néven keresztül is elérhet? ugyanaz a változó, és ha az egyiken keresztül megváltoztatjuk az értékét, akkor mindkett?n nézve megváltozik.

Ahhoz hogy egy változóra a hivatkozást kiszámoljuk, ahhoz használható az **&** operátor, és ahhoz hogy a hivatkozáson keresztül elérjünk egy változót, ahhoz használható a * operátor. Egyszer? példa:

```
int a;
int *mutato;
// Itt a mutato-t egy a-ra valo hivatkozassa tesszuk:
mutato = &a;
// Ettol az a erteke valtozik 5-re
*mutato = 5;
```

Itt egy összefoglaló táblázat ennek a két operátornak a hatásáról.

***a* típusa**	***&a* típusa**	****a* típusa**
int	int*	nincs értelme
int*	int**	int
float	float*	nincs értelme
float*	float**	float

Példa a mutató használatára

Nézzünk egy példát, hogy már ennyi alapján is lehet hasznos a mutató. Tegyük fel, hogy van néhány változónk, és a legnagyobbat ki akarjuk írni, majd a felére csökkenteni. Általánosságban ezt a feladatot két részre bonthatjuk:

1. Kiválasztjuk hogy melyik változót akarjuk módosítani (a legnagyobbat)
2. Azon elvégzünk valamilyen m?veletet (kiírjuk és elfelezzük)

Ennek az általános mintának sok feladat megfelel. Most mindkét lépés elég egyszer?, így akár így is megírhatnánk a kódot:

```
void main()
{
  int a, b, c;
  scanf("%d%d%d", &a, &b, &c);
  // A legnagyobbat modositjuk:
  if(a > b) {
    if(a > c) {
      printf("%d", a);
      a = a / 2;
    } else {
      printf("%d", c);
      c = c / 2;
    }
  } else {
    if(b > c) {
      printf("%d", b);
      b = b / 2;
    } else {
      printf("%d", c);
      c = c / 2;
    }
  }
  // Tovabbiak...
}
```

De azért láthatjuk hogy lényegében ugyanazt a két sort kellett 4-szer leírni. Általános szabály hogy az nem jó a programban ha ugyanaz többször le van írva, mert az általában azt jelenti hogy ha az egyiket módosítani

kell, akkor a többit is, és abból könnyen születnek hibák. Ezért ehelyett jobb megoldás ez:

```
void main()
{
  int a, b, c;
  scanf("%d%d%d", &a, &b, &c);
  int *mutato;
  // A legnagyobbat kivalasztjuk:
  if(a > b) {
    if(a > c) {
      mutato = &a;
    } else {
      mutato = &c;
    }
  } else {
    if(b > c) {
      mutato = &b;
    } else {
      mutato = &c;
    }
  }
  // Majd modositjuk:
  printf("%d", *mutato);
  *mutato = *mutato / 2;
  // Tovabbiak...
}
```

Mutató mint függvényparaméter

Így tehát csak akkor módosítható több helyről egy változó, ha a kódban készítünk egy hivatkozást rá. Ennek a következménye az is, hogy egy függvényben definiált változó csak akkor változhat meg, ha a függvényen belül megváltoztatjuk, vagy ha explicit egy hivatkozást róla átadunk, nem tud meglepni minket, mint egy mutable változó.

```
void f() {
  int x = 4;
  // Ettol nem változhat meg x erteke,
  // g csak egy masolatot kap rola:
  g(x);
  // h-nak egy hivatkozast adunk oda,
  // igy az megvaltoztathatja x erteket:
  h(&x);
}
```

Ez megmagyarázza azt is, hogy miért kell a *printf()*-et és a *scanf()*-et máshogy használni. A *printf()* csak használja az értéket amit odaadunk neki, a *scanf()* viszont meg akarja változtatni a változónk értékét:

```
void main() {
  int n;
  // A scanf()-nek az a celja hogy a
  // valtozonk erteket megvaltoztassa:
  scanf("%d", &n);
  // Vegzunk egy kis szamolast:
  n = n*2;
  // A printf()-nek viszont csak az
  // aktualis erteke kell:
  printf("%d", n);
}
```

Null mutató

A mutatók egy speciális lehetséges értéke a **NULL**. Ez arra használható, hogy ha a mutató változó (még) nem mutat semmire, akkor legyen *NULL* az értéke, és így a program tudja hogy ne próbálja meg ezen keresztül elérni a hivatkozott másik változót, mert nem létezik olyan. Pl.:

```
void otte_tesz(int *p) {
  if(p != NULL) {
    *p = 5;
  }
}
```

Ezért (f?leg persze bonyolultabb programoknál) érdemes lehet a mutató változókat alapból *NULL*-ra inicializálni, hogy ha nem kap értéket, akkor ne próbáljuk használni:

```
void main() {
  int *mutato = NULL;
  int a;
  scanf("%d", &a);
  if(a > 5) {
    mutato = &a;
  }
  otte_tesz(mutato);
  printf("%d", a);
}
```

Mutatók és tömbök

A mutatóra absztrakt módon úgy gondolhatunk, hogy a mutató értéke egy hivatkozás egy másik változóra. A gyakorlatban ez azt jelenti, hogy a mutató értéke az, hogy a másik változó hol található meg a memóriában. Egy tömb elemei mindig egymás mellett találhatók meg a memóriában. Így a mutatók használhatók arra, hogy ide-oda mozogjunk egy tömbben, a + és a - használatával:

```
int v[10] // Egy 10 elemu tomb
int *m = &v[3] // Mutato a tomb 3. elemere
*m = 5 // Modositjuk a tomb 3 elemet
*(m-1) = 8 // Modositjuk a tomb 2 elemet
*(m+3) = 30 // Modositjuk a tomb 6 elemet
```

Ezt nem igazán érdemes használni ilyen rendes tömböknél, mert ha van egy mutatóm a közepére, és onnan megyek jobbra-balra, nem tudom hogy hol van a tömb két vége, és a C nem akadályozza meg hogy tovább menjek valamelyik végénél, és akkor undefined behaviour-ral találkozhatok.

De a karakterláncoknál szokták használni, ott a végén úgyis van egy 0 karakter, ami jelzi a végét, így az meg van oldva. (Ennek megfelel?en általában csak el?refelé szoktak menni a karakterláncokra mutató "http://wiki.math.bme.huchar*"http://wiki.math.bme.hu típusú mutatókkal, mert el?refelé menve tudják hol kell megállni.) Példa, ami kiírja a karakterlánc elemeinek ASCII kódjait:

```
char str[] = "Karakterlanc!";
char *p = &str[0];
while(*p) {  // mivel nem 0 ertek igaznak szamit
  printf("%d ", *p);
  ++p;
}
```

S?t, igazából létrehozás után a C a tömböket és a mutatókat teljesen egyformán kezeli, a tömböt úgy értelmezve mintha a tömb els? elemére mutató mutató lenne. Az egyetlen kivétel hogy a tömb (mint mutató)

nem módosítható, csak az elemei:

```
int tomb[] = {10, 20, 30, 40};
int *p = &tomb[1];
// Ez a tomb elso elemet irja ki:
printf("%d", *tomb);
// Ez a tomb 3. elemet irja ki,
// mert p mar eleve a masodikra mutat:
printf("%d", p[1]);
// Sot ezt is lehet, hogy p is a tomb
// elejere mutasson:
p = tomb;
// Ennek megfelel?en így p a tomb 3.
// elemere mutat:
p = tomb + 2;
// Ezt viszont nem tudom megtenni, a
// tomb valtozo maga nem valtoztathato:
tomb = p;
```

A beépített könyvtárak azon függvényei, amik karakterláncokat várnak paraméternek, ennek megfelel?en azokat *char** típusként várják. Tehát pl. az strcmp(), ami két karakterlánc összehasonlítására használható, két *char** paramétert vár. De mivel ez ekvivalens a tömbbel, azt is nyugodtan lehet neki adni. Példa felhasználás:

```
char sz1[] = "abc\n";
char sz2[] = {97, 98, 99, 10, 0};
if(strcmp(sz1, sz2) == 0) {
  printf("Tenyleg ugyanaz!\n");
}
```

Változók élettartama

Így, hogy van egy plusz módszerünk hivatkozni egy-egy változóra, így fontossá válik tudni a változó élettartamát, hogy meddig létezik egy változó. Erre egy pontos szabály van a C-ben, amit könny? megérteni:

Egy változó annak a blokknak a végéig létezik, amelyikben deklarálták.

(Blokk alatt azt értjük, ami kapcsos-zárójellel van körbevéve.) Nézzük tehát a következ? példát:

```
int g;

void kiir() {
  int i;
  for(i = 0; i < 10; ++i) {
    int j;
    j = i + 1;
    printf("%d ", j);
  }
}

int main() {
  kiir();

  return 0;
}
```

Itt a *g* változó minden blokkon kívül van, így az egész program alatt létezik. (Az ilyent úgy hívjuk hogy globális változó.) Az *i* változó a *kiir()* függvény blokkjában van deklarálva, így az addig létezik amig a *kiir()* függvény fut, azon kívül nem elérhet?. A *j* nev? változó pedig a *for* ciklus blokkjában van deklarálva. Ez azt jelenti, hogy a ciklus mind a 10 futásánál létrejön egy új *j* nev? változó, aminek csak a ciklus adott futásában

van értelme, azon kívül nem létezik ez a változó. Tehát, miel?tt értéket adunk neki, *i+1*-et, el?tte nincs értéke, az egy új változó, akkor is, ha a ciklus el?z? körében volt egy másik *j* nev? változó aminek már volt értéke.

Ennek a mutatók szempontjából az a jelent?sége, hogy az hibának számít, ha egy mutatón keresztül megpróbálunk elérni egy olyan változót, ami már nem létezik. Mutatok egy példát, ami tehát **rossz kód, ne csináljunk ilyet!**

```
int* f(int szam) {
  int dupla = szam*2;
  return &dupla;
}

void main()
{
  int *mutato;
  mutato = f(5);
  printf("%d", *mutato);
}
```

Ez a legegyszer?bb mód ahogy ezt el lehet rontani, de bárhogy próbálunk hivatkozni már nem létez? változóra, az ugyanennyire hiba. Ami még el? szokott fordulni, hogy ugyan az ilyen rossz mutatót nem adjuk vissza a függvényb?l közvetlenül, de letároljuk a mutatót valahova, ahonnan aztán ki van olvasva kés?bb, a függvény futása után.