A C nyelv eredete, fő tulajdonságai

Eredete:

Bell Laboaratories, rendszerprogramozási célra, pl. op. rendszer írására: a UNIX operációs rendszer nagy részét is ebben írták.

K&R C ≡ tradicionális C ≡ az "ős"-C: Dennis M. Ritchie tervezte a nyelvet 1972-ben PDP-11 –re, a könyvet Brian W. Kernighan írta.

<u>Irodalmak:</u>

<u>**A C programozási nyelv - Az ANSI szerinti változat**</u>
B. W. Kernighan - D. M. Ritchie; Műszaki Könyvkiadó, 1995
: Az alábbi könyv magyar kiadása

The C Programming Language Second Edition
Brian W. Kernighan, Dennis M. Rithie; Prentice Hall 1988

: A szabványos nyelvet tartalmazza (majdnem, mert a szabvány rögzítése előtt jelent meg)

<u>**Programozzunk C nyelven!**</u>
BenkőTiborné, Benkő László, Tóth Bertalan; Computer Book, Bp., 1995

The C Programming Language
Brian W. Kernighan, Dennis M. Rithie; Prentice Hall, 1978
Ez csak az ős-C nyelv! Ennek magyar fordítása:

A C programozási nyelv
B. W. Kernighan - D. M. Ritchie; Műszaki Könyvkiadó, 1985

ANSI C: American National Standard X3.159-1989

C - A Reference Manual
Samuel P. Harbison:, Guy L. Steele Jr.; Prentice Hall, 1991
: A szabványos nyelvet, a tradicionálissal való összehasonlítását és precíz magyarázatokat is tartalmaz

Fő tulajdonságai:

- operátor-orientált nyelv: sok, hatékony operátor, pl.
  `++` de `[ ] ( ) = ,` is operátor!
- sok eszköz, operátor, könyvtári függvény ⇒ könnyű benne programot írni
- cím-aritmetika a pointerekkel ⇒ érdekes trükköket lehet ...
- "mellékhatások" ⇒ gyors kódot lehet belőle generálni
- külön fordítás alakja is szabványos, minden gépen létezik ⇒ jól hordozható programokat lehet C-ben írni
- tömör szintaktika ⇒ nehezen olvasható, könnyű hibát véteni, nehéz megtalálni
- előfeldolgozója (preprocesszor) is van, melyben pl. feltételes fordítást, makrókat is használhatunk

+ C++ : a C objektum-orientált kiterjesztése: nagy programok jól struktúrált írásához kiváló
+ Java: C++ újratervezve: platform- (= gép-, op. rendszer-) független: nagyon jól hordozható

⇒ profiknak való, mint Önök ...

Utasítások, operátorok

Néhány általános szabály:

- A kis- és nagybetű különbözik
- Az utasítások formája kötetlen: sorokra tagolás, tabulálás; de
  "a stílus maga a az ember" :
    - tilos tengerikígyó sorokat szülni
    - kötelező a blokkszerkezetnek megfelelő tabulálás
    - kommentezés élet-halál kérdés
- A mellékhatások megzavarhatják az alkotót, hát még az olvasót:

$\Rightarrow$ MISS $\equiv$ Make It Simple and Straight

Utasítások:

Kevés utasítás, de nagyon hatékony, logikus a készlet.

Kifejezés-utasítás: *kifejezés* **;**

Pl.

```
a+b;
a=b;      v1=v2=z-5;
szam=1;   szam++;  z=szam++;  q=++szam;
fun (a, c=b-3);
eljaras(&eredm);
```

Végrehajtása: Kifejezés kiértékelése, az eredmény eldobása.
- nincs értékadó utasítás, de vannak értékadó operátorok!
- a mellékhatások (pl. értékadás, incrementálás) érvényesülnek
- függvény eljárásként is használható, ekkor a visszaadott érték elvész
- van "érték nélküli kifejezés" : nincs értéke: **void**

Blokk: **{** *definíciók és deklarációk* *utasítások* **}**

Pl.

```
if ( a[x] > a[x+1] )      /* cserélni kell */
{  int w;                 /* munkaváltozó a cseréhez */
   w=a[x];  a[x]=a[x+1];  a[x+1]=w;
}
```

IF: *`if ( feltétel )`* *utasítás1*

vagy: *`if ( feltétel )`* *utasítás1*
`else` *utasítás2*

Pl.

```
if (x>y)  min=y;

if (a[x] < min)  min = a[x];        /* minimum */
else  if (a[x] > max) max=a[x];     /* maximum */

if (a[x] < min) {              /* több utasítás */
    min = a[x];                /* minimum és indexe */
    minx = x;
}
else if (a[x] > max) {
        max = a[x];            /* maximum és indexe */
        maxx = x;
    }
```

- a feltétel mindig zárójelben van (más utasításban is)
- a "páratlan" else ág a belső if-hez tartozik:

```
if (felt1)
    if (felt2)  ut1
    else        ut2
```

- üres else ágnál `;` kell:

```
if (felt) ; else  ut
```

SWITCH: `switch (` *egész kifejezés* `) {`
`case` *konstans1* : *utasítás1_1*;
utasítás1_2
...
`case` *konstans2* : *utasítás2_1*;
...
...
`default:` utasítás_d_1;
...
`}`

Végrehajtás: a kifejezés szerinti ág első utasításától kezdve hajtja végre, átcsoroghat a következő ágra, ami tipikus hiba!!!
A kilépéshez **`break`** utasítást kell használni.
Ha egyik konstanssal sem egyezik:
- ha van **`default`** ág, azt hajtja végre,
- ha nincs **`default`** ág, egyiket sem.

Pl. valamilyen adatbázis kezelése:

```
char parancs;
...
scanf (" %c", &parancs);    /* parancs beolv. */
switch (parancs) {
  case 'a':                 /* átcsorog a köv. ágra */
  case 'A':   Add (....);   /* elem hozzáadása */
              break;        /* kilép switch-ből */

  case 'd':
  case 'D':   Delete (....); /* elem törlése */
              break;

  case 'l':
  case 'L':   List (....);   /* listázás */
              break;
    /* ... */
  default :   printf ("\n Hibás parancs\n");
}  /* switch (parancs) */
```

WHILE: `while (` *feltétel* `)` *utasítás*

Pl.

```
x=0;
while (x<n  &&  a[x] != ezkell)
  x++;
```

Végrehajtás: amíg a feltétel igaz, végrehajtja az utasítást

Lásd még: break, continue

DO-WHILE: `do` *utasítás* `while (` *feltétel* `)`

Pl.

```
do{
    x += koz;
    printf ("%10f   %10e", x, f(x));
}while (x<vegertek);
```

A "felesleges" blokk a belső while lehetősége miatt gyakorlatilag mindig kell.

Végrehajtás: utasítás, majd ha a feltétel igaz, újra, mígnem a feltétel hamis lesz.

Lásd még: break, continue

FOR: `for ( ` *inicializáló kifejezés* `;` *feltétel kif.* `;` *léptető kif.* `)` *utasítás*

Pl.

```
for (index=0;  index<vege;  index++)  tomb[index]=0;

for (x=0;  x<n && t[x]!=vegertek;  x++)
    z[x] = fun (x, 2*t[x]);
```

Végrehajtás:

inicializáló kifejezés kiértékelése (és értékének eldobása),
ciklusban:
- feltétel kiértékelése
- ha nem igaz, vége a ciklusnak
- utasítás végrehajtása
- léptető kifejezés kiértékelése

Bármelyik kifejezés hiányozhat. Pl. feltétel nélküli, azaz végtelen ciklus:

```
for (;;) {
    ciklusmag, benne tipikusan break vagy return
}
```

Lásd még: break, continue

2. példa: Tömbben elemek sorrendjének megfordítása:

```
double t [1000];
int n, x1, x2;    /* n  = adatok száma t-ben */
...
for (x1=0, x2=n-1;  x1<x2;  x1++, x2--) {
  double w;
  w=t[x1];  t[x1]=t[x2];   t[x2]=w;
}
```

vagy még C-szerűbb:

```
...
for (x1=0, x2=n-1;  x1<x2;  ) {
  double w;
  w=t[x1];  t[x1++]=t[x2];   t[x2--]=w;
}
```

BREAK: `break;`

Végrehajtás: kilép az őt közvetlenül tartalmazó ciklusból (while, do, for)
ill. switch utasításból

Pl. for ciklussal:

```
int x;
for ( x= ...... ) {
      .....
  hiba = fuggveny (....);
  if (hiba) break;       /* kilép a for ciklusból */
      .....
} /* for x */
```

Tipikus alkalmazás: az első megfelelő szám feldolgozása:

```
for ( x=0;  x<n;  x++) {
  if ( ... ) {        /* ez az első megfelelő elem */
    ...               /* feldologozás */
    printf ( ... );
    break;            /* kilép a ciklusból */
  } /* if */
} /* for x */
```

CONTINUE: `continue;`

Végrehajtás: az őt közvetlenül tartalmazó ciklus magjának további részét átlépi, azaz a feltételvizsgálattal (**`while`**, **`do`**) illetve a léptetéssel (**`for`**) folytatja

Pl.

```
               /* Binom.c -ből vett részlet: */
int n,k;

for (;;) {
  int b;              /* lokális változó e blokkban */
  printf ("\n Írja be N és K értékét"
          " (ha nem szám: vége) :  ");
     if (scanf ("%d%d", &n, &k) <2) break;
     if (n<0)  {
       printf ("\n ***** Hiba: N<0 *****");
       continue;          /* for ciklus folytatása */
     }
     if (k<0) {
       printf ("\n ***** Hiba: K<0 *****");
       continue;
     }
     if (k>n) {
       printf ("\n ***** Hiba: K>N *****");
       continue;
     }

     printf("\n Binom (%d,%d) = %d \n", n, k,
                 Binom (n,k));
}  /* for */
```

RETURN: `return;` `/* ha nincs visszaadandó érték */`

vagy **`return`** *kifejezés* **;**

Végrehajtás: visszatér a függvényből az adott visszatérési értékkel, ha van.
Nem csak a függvény végén lehet.
Ha nincs visszatérési érték, akkor a függvény végén elhagyható.

1. példa:

```
int Binom (int n, int k)
{
  if (k==0 || k==n)  return 1;
  return  Binom(n-1,k-1) + Binom(n-1,k);
} /* Binom() */
```

2. példa:

```
void Kivalasztasos_rendezes (int n,  double a[])
{
   int x1, x2, minx;
   double w;

   if (n<=1) return;  /* nincs mit rendezni */

   for (x1=0;  x1<n-1;  x1++) {

     for (minx=x1,  x2=x1+1;  x2<n;  x2++)
       if (a[x2] < a[minx])  minx=x2;

     w=a[minx];  a[minx]=a[x1];  a[x1]=w;

   } /* for x1 */
                   /* :itt nincs return, bár lehetne */
} /* Kivalasztasos_rendezes () */
```

Operátorok

Sok operátor, sok precedenciaszint, sok szabály
⇒ sok lehetőség, kényelem, hatékony programírás lehetősége
⇒ sok hibalehetőség, hibák nehezebben láthatóak

Precedencia ≡ prioritás és asszociativitás:

Mindkettő két dolgot is jelent: - a kifejezés értelmezése: mi micsoda
- a kifejezés kiértékelési, azaz végrehajtási sorrendje

1. példa:

```
int * ip, it[10];  /* int pointre, tömbje */
int *t[10];  /* pointerek tömbje v. tömb pointere? */
```

Itt `*` és `[]` is operátor:
`[]` : tömb ill. annak indexe operátor (!)
`*` : pointer típus ill. pointer-indirekció; pl.:
`int * ip`;
`ip` egy `int`-re mutató pointer változó.

Mivel `[]` prioritása nagyobb, mint `*` prioritása, az előbbi kif. jelentése:

```
int * (t [10]);    /* pointerek tömbje */
```

2. példa:

```
double * fun (int, char);
```

Mivel a `()` operátor prioritása nagyobb, mint a `*` pointeré, ennek jelentése: `fun` olyan függvény, mely `int` és `char` paramétert használ, és `double`-re mutató pointert ad vissza.

3. példa: a kiértékelés sorrendje:

```
v = a+b/c;    /*  ≡  a + (b/c)   */
```

A kiértékelés sorrendjét itt is zárójelezéssel lehet megváltoztatni, pl:

```
v = (a+b)/c;
```

<u>Asszociativitás</u> ≡ **kötés** ≡ **binding:** al-prioritás az azonos prioritású operátorok között

Pl:

```
a = b / c * d;   /* ≡ a = (b / c) * d  mert ⇒  */
a = b = c;       /* a = (b = c)        mert ⇐  */
a = b * c = d;   /* ≡ a = (b*c) = d    Hibás!   */
a = b *(c = d);  /* rendben */
```

A precedenciát (prioritást és asszociativitást) zárójelezéssel lehet felülírni.

Rövidrezárt kiértékelés: `&&` (loikai ÉS) illetve `||` (logikai VAGY) operátornál:

- az első (bal) operandus kiértékelése,
- ha a logikai kifejezés értéke ezzel eldőlt, akkor a második (jobb) operandust nem is értékeli ki

1. példa: lineáris keresés tömbben:

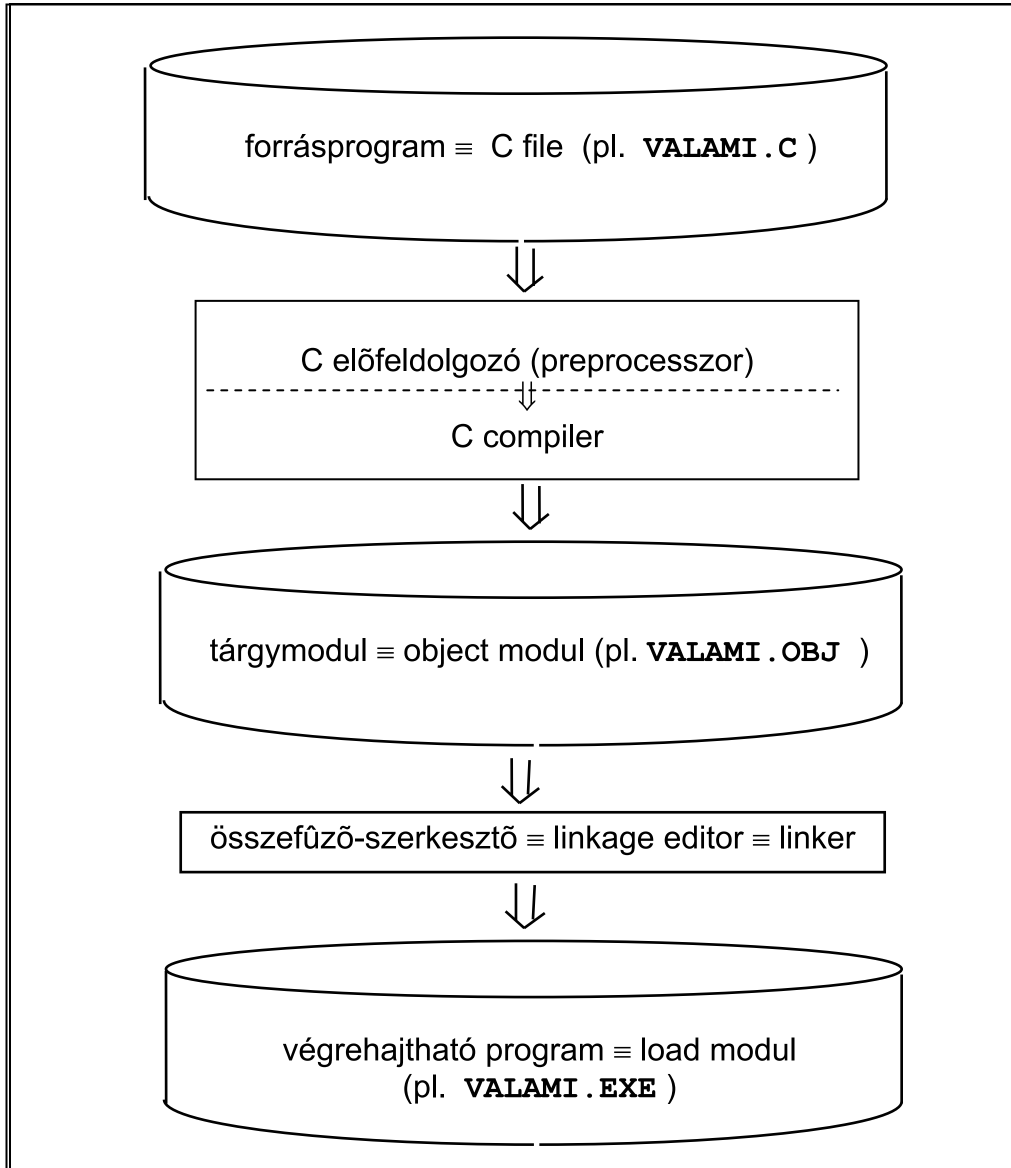
```
for (x=0;  x<n  &&  t[x] != keresett;  x++) ;
```

2. példa: keresés láncolt listában:

```
for (p=eleje;
     p != NULL  &&  p->kulcs != keresett;
     p = p->kovetkezo) ;
```

A preprocesszor

C nyelvű programok feldolgozása:



A preprocesszor és a compiler általában egyetlen program, de esetleg külön is használható.
Pl. a GNU C++ compilernek csak a preprocesszora fut le, ha a **`-E`** opciót használjuk:

```
gcc -E valami.c -o valami.pp
```

Ekkor a **`valami.c`** program preprocesszált szövege kerül **`valami.pp`** -be.

Az előfeldolgozó működése

Az előfeldolgozó, amit gyakran makrofeldolgozónak, vagy makroprocesszornak is neveznek (de nem az), egy szövegbehelyettesítő program, amely feldolgozza a neki szóló parancsokat és szöveget állít elő, melyet a C fordító fordít le.

Az előfeldologozó parancsok alakja:

```
#parancs  argumnetum1 argumentum2 ... egyéb szöveg
```

ahol a **#** előtt, valamint a **#** a és a **parancs** között csak szóközök és tabulátor jelek lehetnek.

Fontosabb preprocesszor parancsok:

#include: file belevétele

```
#include <file.kit>
#include "file.kit"
#include preprocesszor_tokenek
```

:teljes file behelyettesítése az #include -ot tartalmazó sor helyére;

<file.kit> :a fordítóhoz tartozó szabványos include könyvtárban keres először a preproceszor ⇒ a könyvtári un. header file-okhoz használjuk

"file.kit" :a saját, pl. aktuális könyvtárban keres először ⇒ a saját header file-okhoz szoktuk használni

Hogy utána hol keres, az megvalósításfüggő, általában a C fordító opciójaként beállítható.

#include -ok tetszőleges mélységben egymásba ágyazhatóak, ha a megvalósítás ezt nem korlátozza.
Pl.:

```
afile.c:                bfile.c:                  cfile.h
--------                --------                  -------
a1a1                    b1b1b1                    cccccccc
#include "bfile.c"      #include <cfile.h>
a2a2                    b2b2b2
```

Az előfeldolgozás eredménye, a C compiler ezt kapja:

```
a1a1
b1b1b1
cccccccc
b2b2b2
a2a2
```

#define: makró definiálása

```
#define makronév
#define makronév  makrotörzs
#define makronév(paraméter1,paraméter2...) makrotörzs
```

:a makronév helyére behelyettesíti a makrotörzset, tokenenként, a makrotörzsben a paraméterek helyére az aktuális paraméterek kerülnek

Pl.

Eredeti szöveg	Előfeldolgozás után
`#define tablahossz 100`	<semmi>
`...`	`...`
`int tabla [tablahossz];`	`int tabla [100];`
`...`	`...`
`#define min(a,b) ((a)<(b) ? (a):(b))`	<semmi>
`...`	`...`
`x=min(z-1,y)`	`x=((z-1)<(y) ? (z-1):(y))`
`...`	

A "felesleges" zárójelek a precedencia miatt kell**het**nek, ha az argumentumok kifejezést tartalmaznak, vagy a makrót kifejezésben használjuk!

Szabályok:
- Nincs szünet (szóköz, tabulátor) a makronév és a `(` között a makrodefinícióban, mert az jelzi a makrotörzs elejét, de lehet a hívásban.
- A definícióban a paraméterlista lehet üres is, ekkor a hívásban is kell üres `()` pár, ezzel függvényhívást mímel. Pl.:

```
#define valami() tozse pl. ez
     ....
.... valami() ... /* makro hívása: kell az üres () */
```

- A makro-kiterjesztés után az így kapott szöveget ismét átnézi a preprocesszor, és ha talál benne makróhívást, azt szintén feldolgozza, stb. De nincs makro-kiterjesztés a makrotörzsben ill. a makroparaméterekben, azaz nem lehet makrodefiníciót ill. makroparamétert makróból generálni.

- Rekurzív makrohívás második szintjét már nem helyetesíti be a preprocesszor. Pl.:

```
#define m1   m1-ben m2 hívása
#define m2   m2 törzse
.... m1 ....    ⇒  .... m1-ben m2 törzse hívása ....
```

- Makrodefiníció tipikusan egyetlen sor, de ha több sorban folytani akarjuk, akkor a folytatott sor végén, pontosan az újsor karakter előtt **** karakter kell, ezt és a követő újsor jelet az előfeldolgozó a makrók feldolgozása előtt lenyeli, azaz az ilyen sorokat egyesíti. Pl.

```
#define mnev(par1,par2) törzs első sora, végén  \
                 második sor, ezt is folytatjuk: \
                 utolsó sor
```

A **** a sor végén általában is használható egy sor folytatására, nem csak makrodefinícióban. Pl.

```
valami = "string eleje, de fol\
ytatható új sorban";
```

Ez ugyanaz, mint:

```
valami = "string eleje, de folytatható új sorban";
```

vagy mint:

```
valami = "string eleje, de fol"
           "ytatható új sorban";
```

#undef : makródefiníció törlése

```
#undef makronév
```

Hatására a makrodefiníció törlődik a preprocesszorból, utána újabb makró definiálható ugyanilyen névvel. Ha nem törölnénk a makrodefiníciót, akkor csak ugyanilyen törzsű makrót definiálhatnánk ilyen névvel.

Pl.:

```
/* Hibaüzenetet kiíró makró: */
#define HIBA { printf("Hiba a(z) %s függvényben", \
        FUGGVENY_NEV);  abort(); }
...
     /* Függvény nevét definiáló makró: */
#define FUGGVENY_NEV  "Funny"

void Funny (int x)
{    ...
     if (....) HIBA;          /* 1 */
     ...
     switch (....) {
       case 'D': ...
       case 'Q': HIBA;        /* 2 */
                 break;
       ...
} /* Funny() */
#undef FUGGVENY_NEV


     /* Másik függvénynevet definiál: */
#define FUGGVENY_NEV  "Masik"

double * Masik (char * cp)
{
     ...
     if (...) HIBA;           /* 3 */
     ...
}
```

A HIBA hívásokból generált sorok rendre:

```
{ printf("Hiba a %s függvényben", "Funny");  abort(); }

{ printf("Hiba a %s függvényben", "Funny");  abort(); }

{ printf("Hiba a %s függvényben", "Masik");  abort(); }
```

#ifdef : feltételes fordítás makro létezésétől függően

```
#ifdef makronév                  vagy:   #ifdef makronév
  ezt gen. ha definiált                     ezt gen. ha definiált
#else                                    #endif
  ezt gen. ha nem definiált
#endif
```

A preprocesszor az

`#ifdef` és `#else` illetve `#ifdef` és `#endif`

közötti részt, ami több sor is lehet, csak akkor generálja, ha a makró definiálva volt, egyébként

az `#else` és az `#endif`

közötti sorokat generálja, vagy semmit, ha nincs ilyen rész.

#ifndef: feltételes fordítás makro nem-létezésétől függően

```
#ifndef makronév
     ...
```

esetén pedig csak akkor, ha NEM volt definiálva e makró.

1. példa: Egyes dolgokat a program belövésekor végre akarunk hajtani, de a végleges változatban már nem:

```
...
#define TESZTELES   /* ha már nem tesztelés, akkor pl.
                    /* -ot kell tenni a sor elejére */

...    /* : mindig szükséges rész */
#ifdef TESZTELES
printf ("\n Most x = %d",  x); /* Csak ha még teszteljük*/
#endif
...    /* : további, mindig szükséges rész */
#ifdef TESZTELES
if (z<0 || z>=n) {            /* z értéke rossz */
     printf ("\nBaj van:  z = %d", z);
     abort();                 /* McGouglas */
}
#endif
```

2. példa:

Include-olási fa: f3.c ⇒ f1.h ⇒ f0.h
⇒ f2.h ⇒ f0.h

<u>f0.h file:</u>

```
#ifndef s1definialt
#define s1definialt /* csak makronevet definiál, nincs
               makrotörzs, azaz behelyettesítendő szöveg */
typedef struct {    /* egy struktúra típus definiálása */
               ....
} s1;               /* s1 a típusnév lett */
#endif
```

<u>f1.h file:</u>

```
#include "f0.h"
int g (s1 *p)  /* függvény, mely használja s1 típust */
{  ...  }
```

<u>f2.h file:</u>

```
#include "f0.h"
int ff (s1 *p)  /* másik függvény, mely használja s1-et */
{  ...  }
```

<u>f3.c file:</u> Ez közvetve másodszor is #include-olja f0.h-t, de abban **s1** definiálása már nem történik meg újra

```
#include "f1.h"
#include "f2.h"
int main () {
     s1 v1,v2;
          ...
     a=g(&v1) + ff(&v2);
          ...
}
```

f3.c a behelyettesítések után:

```
typedef struct {          ⇐ f1.h -beli #include-ból keletkezett
               ....
} s1;
int g (s1 *p)             ⇐ f1.h szövege
{  ...  }
int ff (s1 *p)            ⇐ f2.h szövege, előtte nincs typedef ....
{  ...  }
int main () {             ⇐ f3.c szövege
     ...
```

#if: feltételes fordítás

```
#if konstans-kifejezés        vagy:     #if konstans-kifejezés
  generálja, ha != 0                        generálja, ha != 0
#else                                     #endif
  generálja, ha == 0
#endif
```

Ha a konstans-kifejezés (amelyet a preprocesszor ki tud értékelni long int vagy unsigned long int típusú adatként, azaz mely nem függ futási időbeli értéktől)
nem 0, akkor az 1. sor-csoportot dolgozza fel,
egyébként, ha van, akkor a 2. sor-csoportot. Pl.

```
#define  hossz ....                     :egy előző definíció
     ....
#if hossz<1024            /* túl kicsi */
  #undef hossz
  #define hossz 1024      /* így legalább 1K lett */
#endif
```

Több, egymásba ágyazott **`#if`** helyett egyszerűbb alak:

```
#if konstans-kifejezés-1
     1. sor-csoport
#elif konstans-kifejezés-2
     2. sor-csoport
#elif konstans-kifejezés-3
     3. sor-csoport
#elif konstans-kifejezés-4
     4. sor-csoport
...
#else                              /* Ez a rész ... */
     utolsó sor-csoport            /*    elhagyható */
#endif
```

:csak az egyik sorcsoportot dolgozza fel, amelyiknek a feltétele először teljesül, vagy csak az **`#else`** utánit, ha van.

defined operátor:

```
#if defined makronév          /* ugyanaz mint: */
#ifdef makronév

#if !defined makronév         /* ugyanaz mint: */
#ifndef makronév
```

#error: fordítási hibaüzenet generálása

```
#error preprocesszor tokenek
```

Lehetővé teszi hibaüzenetek generálását, ha valami hibás beállítást tapasztalunk a preprocesszálás során. Ez az üzenet megjelenik a fordítóprogram szintaktikai hibaüzenetei között. Pl.

```
#include "definitions.h"
  /* ennek pl. definiálnia kell egy megfelelő értékkel
     a MERET makrót */
#ifndef MERET                  /* nem definiálta? */
     #error "MERET" nincs definiálva
#elif (MERET % 256) != 0       /* jó az értéke? */
     #error "MERET csak 256 egész többszöröse lehet"
#endif
```

`#line, #pragma:` lásd könyv (kevésbé fontosak)

Az előfeldolgozás sorrendje:

1. Trigraph -ok behelyettesítése:

`??=`	⇒	`#`	`??/`	⇒	`\`	`??'`	⇒	`^`
`??(`	⇒	`[`	`??)`	⇒	`]`	`??!`	⇒	`|`

2. Sorvégi **`\`** és az azt követő újsor karakter törlése
3. A kommentek helyére egy-egy szóköz kerül
4. A szöveg tokenekre (lexical unit) bontása
5. Preprocesszor parancsok végrehajtása
6. Makrók behelyettesítése
7. Escape sorozatok behelyettesítése karakter- és string-konstansokban
8. Egymást követő string-konstansok egyesítése

További példa **#if** használatára:

C-ben nem lehet a kommenteket egymásba ágyazni:

```
/*  külső komment eleje
     ...
/* belső komment eleje                ⇐ már kommentben van, nem számít
     ...
   belső komment vége:  */            ⇐ ez a külsőt fejezi be!!!
     ...
   külső komment vége */              ⇐ hiba!
```

Helyette **#if** -eket használunk a külső szinteken, akár egymásba ágyazzva:

```
#if 0
   külső komment eleje
     ...
/* belső komment eleje
     ...
   belső komment vége:  */
     ...
   külső komment vége

#endif
```

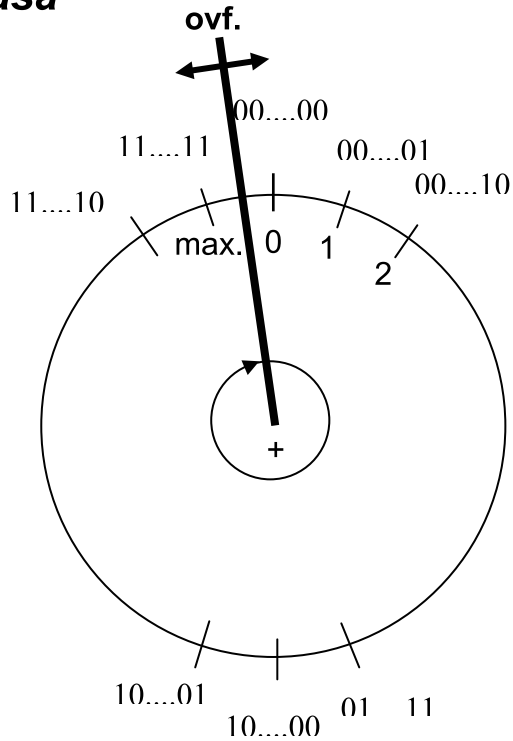
+ Makró argumentuma stringként is behelyettesíthető, ld. a string típusnál (c-ea-2.doc).
+ További, kevésbé fontos preprocesszor parancsok: **#line, #pragma**

Alapvető adattípusok

Egészek bináris ábrázolása

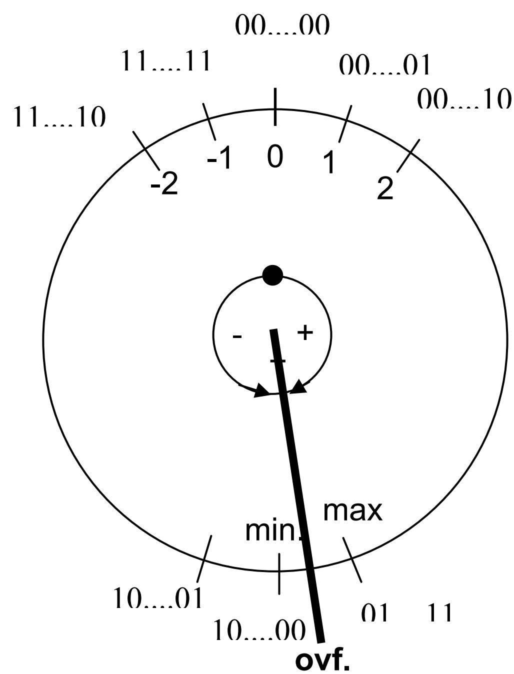
Előjel nélküli egész,

pl.: **`unsigned int`**



Előjeles egész,

pl.: **`int`**



C-ben sok adattípus áll rendelkezésre és ezek jól reprezentálják a processzor saját belső adatábrázolását.

Adattípus két dolgot jelent:

- értékkészlet ≡ milyen értékek tárolhatók benne,
- műveletek ≡ milyen műveletek végezhetők vele (nem csak aritmetikai!)

C típusok:

- void
- skalár:
  - aritmetikai:
    - egész:
      - integer
      - karakter
      - felsorolás
    - lebegőpontos
  - mutató
- függvény
- union
- összetett:
  - tömb
  - struktúra

Egész típusok

Az értékkészleteket a `<limits.h>` file adja meg az adott implementációhoz.

Előjeles egész

Számábrázolás: kettes complemens kód (szinte mindig):

Értékkészlet:	**minimum**	**-1**	**nulla**	**1**	**maximum**
bináris alak:	10 ... 00	11...11	00...00	00...01	011...11
érték n biten:	-2^{n-1}	-1	0	1	$2^{n-1}-1$

Típus nevek, minimális és maximális értékek:

	minimális hossz [bit]	típus-megadás alakja	minimális érték <limits.h> -ban	maximális érték <limits.h> -ban
rövid	**16**	**short** **short int** **signed short** **signed short int**	**SHRT_MIN**	**SHRT_MAX**
normál	**16**	**int** **signed int**	**INT_MIN**	**INT_MAX**
hosszú	**32**	**long** **long int** **signed long** **signed long int**	**LONG_MIN**	**LONG_MAX**

Szám-konstans alakja:

Típus	**Számrendszer**	**Példák**
int	decimális	**0 123 -33**
	oktális	**012 0177777**
	hexadeimális	**0x1a 0x7fff 0xAa1bB**
long		**-″-L -″-l** vagy ha az érték olyan nagy

Előjel nélküli egész

Számábrázolás: kettes számrendszerben

Értékkészlet:	**minimum ≡ 0**	**1**	**maximum**
bináris alak:	00 ... 00	00...01	11...11
érték n biten:	0	1	2^n-1

Típus nevek, minimális és maximális értékek:

	minimális hossz [bit]	típus-megadás alakja	minimális érték	maximális érték <limits.h> -ban
rövid	**16**	**unsigned short** **unsigned short int**	**0**	**USHRT_MAX**
normál	**16**	**unsigned** **unsigned int**	**0**	**UINT_MAX**
hosszú	**32**	**unsigned long** **unsigned long int**	**0**	**ULONG_MAX**

Szám-konstans alakja:

unsigned	<mint **int>u** <mint **int>U**
unsigned long	<mint **int>ul** <mint **int>UL** <mint **int>lu** <mint **int>lU** ... vagy ha az érték olyan nagy

Automatikus ábrázolási mód, ha nincs megadva U vagy L:

amelyikbe a szám-konstans előbb belefér:

1. **int**
2. **unsigned int** :csak oktális és hexa esetén
3. **long int**
4. **unsigned long int**

Pl. (feltéve, hogy int 16 bites):

```
int :               0  1 -1  32767  0xff  077777
unsigned int  :     32768  0123456  0xffff
long int :          66000  0x10000  0x7fffffff
unsigned long :     0x80000000  3123456789
```

Számításnál a ... **short** típusú értéket előbb automatikusan mindig ... **int** típusúvá alakítja, és azzal számol.

Karakter típusok (szinte mint egy egész típus)

Ábrázolás: tipikusan 1 darab 8 bites byte -on.

Hogy egy egyszerű `char` előjeles-e, az nem definiált, a megvalósítás lehet előjeles és előjel nélküli is, sőt keverheti is a kettőt :egyszer így, máskor úgy!

Alak	Min. értékkészlet	Értékhatárok <limits.h> -ban
signed char	-128 ... 127	**SCHAR_MIN ... SCHAR_MAX**
unsigned char	0 ... 255	**0 ... UCHAR_MAX**
char	akár -128 ... 127 akár 0 ... 255	**CHAR_MIN ... CHAR_MAX**

Karakterkonstansok alakja, speciális értékek:

Karakter konstans (literál)	Alakja	Jelentése
egyszerű karakterek:	**’a’**	kis a betű
	’:’	kettöspont
spec. karakter escape kdja	**’\’’**	aposztróf : ’
	’\a’	alarm ≡ hangjelzés
	’\b’	backspace ≡ visszatörlés
	’\f’	form feed ≡ lapdobás
	’\n’	new line ≡ újsor jel
	’\r’	carriage return ≡ kocsi vissza
	’\t’	horizontális tabulátor
	’\v’	vertikális tabulátor
	’\\’	backslash ≡ \
	’\”’	idézőjel
	’\?’	kérdőjel
numerikus escape karakterek	**’\0’**	nulla értékű karakter
	’\10’	oktális 10 ≡ 8 decimális ért. kar.
	’\x10’	hexadecimális kar. ≡ 16 dec. ért.

Számításnál a ... **char** típusú értéket előbb automatikusan mindig ... **int** típusúvá alakítja, és azzal számol.

Lebegőpontos típusok

Típus	Konstans alakja	Min. abs. érték	Max. abs. érték	pontoság [dec. jegy]

`float`	`12.3f` `0.12F`
	`12.F` `.5f`
	`1E-3f` `1.8e5f`
		`FLT_MIN`
		`<= 1e-37`
			`FLT_MAX`
			`>= 1e37`
				`FLT_DIG`
				`>= 6`

`double`	`12.3` `0.12`
	`12.` `.5`
	`1E-3` `1.8e5`
		`DBL_MIN`
		`<= 1e-37`
			`DBL_MAX`
			`>= 1e37`
				`DBL_DIG`
				`>= 10`

`long double`	`12.3L` `0.12l`
	`12.l` `.5L`
	`1E-3l` `1.8e5L`
		`<= mint`
		`double`
			`>= mint`
			`double`
				`>= mint`
				`double`

⇒ az alapvető “valós” típus a **`double`**

Aritmetikai típusok konverziója

Egy-operandusú konverziók (unary conversions):

Pl. értékadásnál: `ebbe = ezt;`

aktuális paraméter-átadásnál:

```
/* Függvény definíciója: */
valami Fuggv (ebbe formalispar)
{     ....
}
...
Fuggv (ezt);
```

Alapelv : érték megőrzése, ha lehet
Túlcsordulás esetén a kapott érték elvileg definiálatlan, azaz bármi lehet, kivéve néhány egész konverziót, lásd alább.

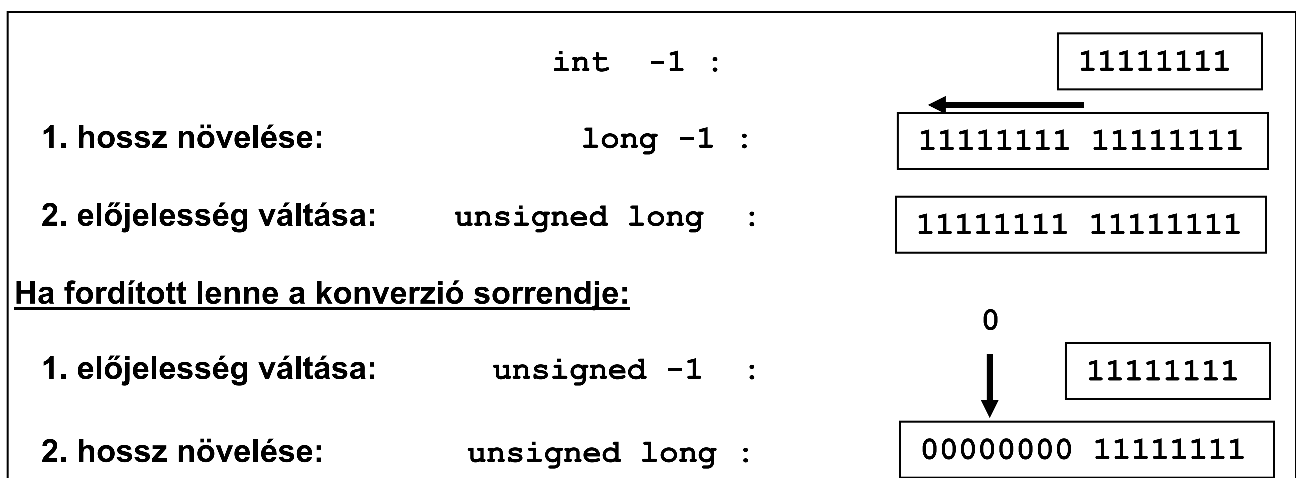
Egészből egészbe: NINCS TÚLCSORDULÁS-JELZÉS !!!

Túlcsorduláskor egyes esetekre van szabály, másokra nincs, lásd a táblázatot.

A gyakorlatban a legtöbb gép kettes komplemens kódban dolgozik, ekkor az eredményt, vagy annak alsó n bitjét - ahol n a célterület bithossza - teszi a célterületre, ami matematikailag azt jelenti, hogy az eredmény a (helyes modulo 2^n) értéke lesz.
(Ha nem kettes komplemens kódban dolgozik, akkor az előjeles eredmény értéke túlcsorduláskor definiálatlan, az előjel nélkülié modulo 2^n.)

Ha a konverziónál a hossz is nő, és az "előjelesség" is változik, akkor először az eredeti előjel szerint növeli a hosszat, utána váltja előjelesre ill. előjel nélkülire. Pl.

```
unsigned long ul;  ul = -1;  /* =ULONG_MAX, mert ugyanaz, mint:*/
                    { long L=-1;  ul=L; }
  /* tehát nem: */  { unsigned u=-1;  ul=u; }  /* =UINT_MAX */
```



Miből	Mibe	Eredmény
`char, short`	`int`	mindig, minden művelet előtt
rövidebb `int` pl. `short` `int`	hosszabb `int` ⇒ `int` ⇒ `long`	O.K.
rövidebb `unsigned` pl. `unsigned short` `unsigned`	hosszabb `unsigned` ⇒ `unsigned` ⇒ `unsigned long`	O.K.
hosszabb `int`	rövidebb `int`	túlcsordulhat (felső bitek elvesznek)
hosszabb `unsigned`	rövidebb `unsigned`	túlcsordulhat (felső bitek elvesznek)
rövidebb `+ int`	`unsigned`	O.K.
`- int`	`unsigned`	modulo 2^n, azaz **ebbe = ezt +** 2^n (**> ~_MAX** !!!)
`unsigned`	ugyanakkora `int`	túlcsordulhat
`unsigned`	hosszabb `int`	O.K.

Pl.:

```
int x =-1;
unsigned u1;   u1 = x;     /* u1 = UINT_MAX lett !!! */
```

<u>Lebegőpontosból lebegőpontosba:</u> van túlcsordulás-jelzés

Miből	Mibe	Eredmény

rövidebb lebegőpontos pl.:
`float` ⇒
`double` ⇒
	hosszabb lebegőpontos
	`double`
	`long double`
		O.K.

hosszabb	rövidebb	túlcsordulhat, pontosság csökkenhet

<u>Egész - lebegőpontos között:</u> van túlcsordulás-jelzés

Miből	Mibe	Eredmény
egész	lebegőpontos	ha lehet, a pontos érték megtartásával, ha nem: a legközelebbi két érték egyikére, ha nem lehet: túlcsordul
lebegőpontos	egész	törtrész elhagyásával, túlcsordulhat

Két-operandusú konverziók (binary conversions)

Két-operandusú operátoroknál, pl.

```
int      x;
unsigned u;
long     L;
x = ...;  u = ...;   L = x+u;
```

Itt az összeadáshoz két-operandusú, az értékadáshoz egy-op. konverzió kell.

A konverzók sorrendje rendre:

0. Ha egyik tömb, vagy függvény, akkor mutatóvá konvertálódik, és nincs további konverzió
1. (Csak aritmetikai típusúak lehetnek)
   Egy-operandusú konverzió a hossz növelésére
2. A két operandus azonos típussá alakítása a következő sorrend szerint:

Egyik operandus	Másik operandus	Közös, új típus
`long double`	bármi	`long double`
`double`	bármi	`double`
`float`	bármi	`float`
`unsigned long`	bármi	`unsigned long`
`long`	bármi (`int, unsigned`)	`long`
`unsigned`	bármi (`int`)	`unsigned`
`int`	bármi (`int`)	`int`

(Mint már tudjuk, short ill. char típusú értéket előbb mindig int típusúvá alakítja, azzal számol.)

```c
/*============ 1. mintaprogram =============*/

#include <stdio.h>    /* file-kezelő függvények könyvtára */
#include <limits.h>   /* határértékek, pl. INT_MAX */

int main (void)
{
     int n,f,x;

     printf ("\n Irj be egy egész számot : ");
     scanf ("%d", &n);             /* beolvasás és konvertálás */

     for (f=1, x=2;  x<=n;  x++)
          if ( f > INT_MAX/x) {    /* f*x már túlcsordulna */
               fprintf (stderr, "\n Túlcsordul, a szám"
                                " legfeljebb %d lehet\n", x-1);
               return 1;           /* hibakód */
          }
          else f *= x;             /* biztosan helyes */

     printf("\n %d faktoriális = %d\n", n, f);

     return 0;      /* O.K. */
}
```

Néhány rossz megoldás a túlcsordulás vizsgálatára:

```c
if (f*x > INT_MAX) ....
```

Rendkívül naív megoldás, egy int nem lehet nagyobb, mint INT_MAX !!!

```c
if (f*x) <0) ....
```

Kicsit jobb, mert ha a szorzat negatív, akkor biztosan túlcsordulás volt, de ha pozitív, akkor nem biztos, hogy nem. Például 16 bites egészekkel 256*257 eredménye 256, ami pozitív, pedig nyilván túlcsordult!

```c
if ((long)f * x >INT_MAX) ...
```

Ez azt jelenti, hogy ha a szorzás egyik operandusát `long int`-é konvertáljuk, akkor először a másikat is azzá alakítja, a szorzást ezekkel végzi, az eredmény is `long`. Így az összehasonlítás előtt `INT_MAX` -ot is először long-á alakítja. A szorzat viszont csak akkor lesz biztosan helyes, ha a `long` legalább kétszer olyan hosszú, mint az `int`, ez pedig egyáltalán nem biztos.
További gond: ha ugyanezt a feladatot `long f` és `x` értékekkel is meg akarjuk oldani, akkor már nincs még hosszabb integer típus, amit bevethetnénk.

A négy alapművelet:

`* / %` (`/` osztás, `%` modulo)
`+ -`

Casting operátor: **(*típus*)**

Hatása:
- aritmetikai típusok között: egy-op. konverzió
- más típusok között: az utána álló értéket adott típusúnak tekinti, DE NEM KONVERTÁL SEMMIT, azaz a fordítóprogram típus-ellenőrzését kapcsolja csak ki az adott értékre.

Pl. aritmetikai típusokkal:

```
int x = 20000, y = 8000;
long L;
double d;

L = x * y;           /* nem szerencsés: a szorzást int-ekkel
                        végzi, túlcsordulhat */
L = (long) x * y;    /* x-et long típusúvá alakítja, így y-t is,
                         asz orzást long-okkal végzi */
d = x/y;             /* az osztást int-ekkel végzi az eredmény
                        is csonkított int = 2, ezt konvertálja
                        double-é, d=2 lett */
d = (double)x / y;   /* x-et double-é teszi, így y-t is, az
                        osztás eredménye 2.5 */
```

Pre-increment, -decrement: **++** *op* **--** *op*

1-el megnöveli, ill. lecsökkenti az utána álló változót, az eredmény (azaz az egész kifejezés értéke) a megváltozott érték. Pl.:

```
int x=0, y;    /* x=0   y= ??? */
y = ++x;       /* x=1   y=1    */
```

Post-increment, -decrement: *op* **++** *op* **--**

veszi a változó régi értékét, az eredmény (azaz az egész kifejezés értéke) ez a régi érték, majd valamikor később 1-el megnöveli, ill. lecsökkenti az előtte álló változót. Pl.:

```
int x=0, y;    /* x=0   y= ??? */
y = x--;       /* x=-1  y=0    */
```

Hogy pontosan mikor végzi el a növelést ll, csökkentést, csak annyiban definiált, hogy a következő sorrend-határig.

Sorrend-határ (sequence point):

A program végrehajtásának azon pontja, melynél minden előző mellékhatás végrehajtása be kell, hogy fejeződjék és egy későbbi mellékhatás végrehajtása sem kezdődhet el.
Ezek a határpontok:

- utasítás-határ,
- teljes kifejezés kiértékelésének vége,
- vessző operátor,
- függvény aktuális paraméterének meghatározása,
- `&&` (logikai ÉS) és `||` (logikai VAGY) bal operandusa után,
- `?:` (feltételes kifejezés) első operandusa után,
- `,` (vessző operátor) első operandusa után

Pl.

```
int x=1, z;
z = ++x * ++x;  /* z értéke 6 és 9 is lehet, x biztosan 3 */
```

Egy feladat:

Készítse el a következő Pascal programrészlet C-beli megfelelőjét úgy, hogy minden **TOL** és **IG** értékre jól működjön:

```
var   TOL, IG, x :integer;
...
TOL := ...;    IG := ...;

for x:=TOL to IG do valami;
```

Naiv megoldás:

```
int TOL, IG, x;
...
TOL = ...;   IG = ...;

for (x=TOL;  x<=IG;  x++)  valami;
```

Ha ezt sikerült csak összehozni, akkor először készítse el a következőnek a jó változatát! Pascal eredeti:

```
for x:=TOL downto 0 do valami;
```

C-ben ezt oldja meg **unsigned** típussal! Ez nem jó:

```
unsigned x, TOL = ...;
...
for (x=TOL;  x>=0;  x--)  valami;
```

Ez jobb:

```
for ( x=TOL;  ;  x--) {
      valami;
      if (x == 0) break;
}
```

Az eredeti feladat jó megoldása, **INT_MAX** -ra is gondolva:

```
for (x=TOL;  x<=IG;  x++) {  /* a feltétel a TOL>IG
                                estére kell */
      valami;
      if (x == IG) break;
}
```