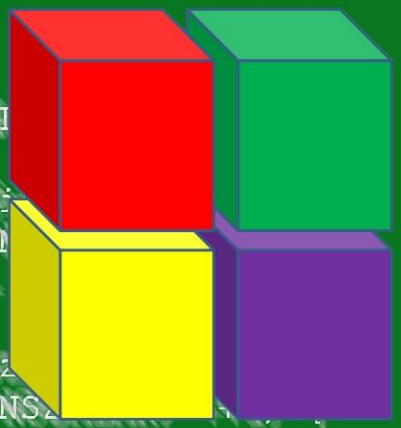
```
#include <stdio.h>
#include <threads.h>
#include <time.h>

#define NSZALAK 7

static int szalSorszam[NSZALAK];

int SzalFuggveny(void * data) {
  struct timespec ido;
  printf("%d. szal inditasa...\n", *(int*)data);
  ido.tv_sec = 3;
  thrd_sleep(&ido, NULL);
  printf("%d. szal felebredt...\n", *(int*)data);
  return 0;
}

int main(void) {
  thrd_t szalID[NSZAL
  // Száladatok inici
  for (int i=0; i < N
    szalSorszam[i] =

  // NSZALAK számú sz
  for (int i=0; i < NS
    if (thrd_create(szalID+i,
          SzalFuggveny, &szalSorszam[i]) != thrd_success) {
      printf("%d. szal sikertelen inditasa.\n", i+1);
      return i+1;
    }
  }

  // Várakozás minden szál leállására
  for (int i=0; i < NSZALAK; ++i)
    thrd_join(szalID[i], NULL);

  return 0;
}
```



TÓTH BERTALAN

A C99 ÉS A C11 SZABVÁNYOK ÚJ LEHETŐSÉGEINEK ÁTTEKINTÉSE

Tóth Bertalan:

A C99 és a C11 szabványok új lehetőségeinek áttekintése

– 2015–

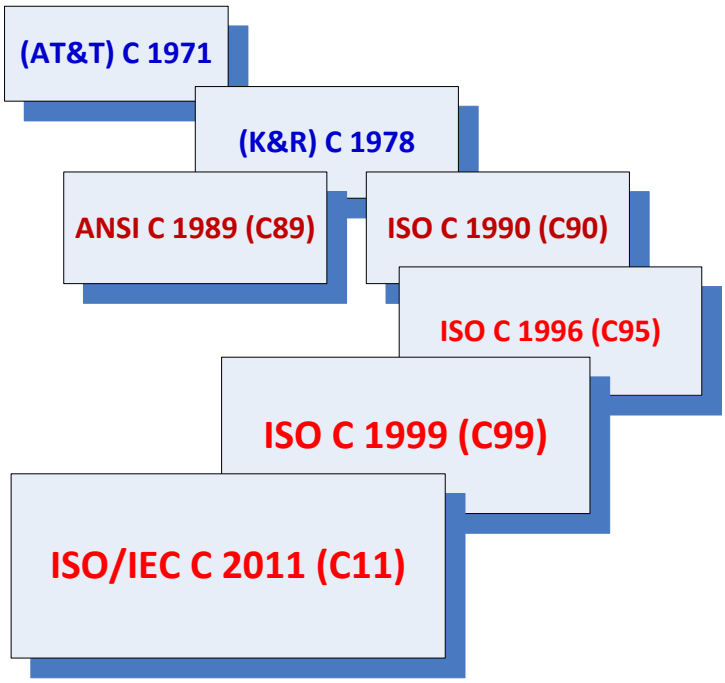
Lektorálta: Juhász Tibor

Tartalom

A C99 és a C11 szabványok új lehetőségeinek áttekintése

Bevezetés

Az 1999. december 1-én megjelent C szabvány **ISO/IEC 9899:1999** (röviden **C99**) nyugodtan nevezhető merésznek, sőt vakmerőnek. Ennek egyik oka, hogy feladta a nem sokkal korábban szabványosított C++ nyelvvel való kompatibilitást, a másik pedig az, hogy egy sor fejlesztői igényt figyelembe vett, azonban távolról sem mindet.



Az alábbiakban témakörökre bontva bemutatjuk a C99 nyelvet. A bemutató során építünk az ANSI C89 (ISO C90) nyelv ismeretére, így azok elemeit már nem részletezzük.

A C szabvány legutolsó **ISO/IEC 9899:2011** (röviden **C11**) változata a 2011. december 8-én látott napvilágot. Míg a C99 nyelv alapvetően szakított a C++ nyelvvel, addig a C11 nyelv több megoldása ismét a C++ irányába viszi a C nyelvet. További szempont volt a C nyelv újabb dialektusának kialakításakor a biztonságos programkészítés segítése, valamint bizonyos C99 megoldások javítása. A nyelv legfontosabb része azonban a többszálúság szabványos, nyelvi szinten történő támogatása. A C11 szabvány újdonságait a C99 nyelv bemutatását követő 7. fejezetben foglaltuk össze.

Valószínűleg a C++ nyelvtől való eltávolodása az oka annak, hogy mind a mai napig nem született olyan C fordító, amely teljesen megfelelne a C99 szabvány előírásainak, nem is beszélve a C11 nyelv támogatásáról. Néhány nagyobb fordítófejlesztő (*Microsoft*, *Borland*/*Embarcadero, IBM*, *Sun*) csupán a könnyen megvalósítható részek támogatását tűzte ki célul, és inkább az új C++ szabványokra koncentrálnak. Csak az ingyenes *GCC* és *Clang* fordítók fejlesztőinél látható a törekvés a mind teljesebb C99 fordítóprogram elkészítésére.

A **C11** szabvány bizonyos elemeinek támogatására ugyancsak a *GCC* és a *Clang* valamint az *IBM XL C/C++* fordítóprogramok legutóbbi változataiban láthatunk példát. Napjainkban (*2015*) a C99 és C11 nyelvek legteljesebb megvalósítását - a szintén ingyenes – *Pelles C* fordítóban találjuk meg.

Annak ellenére, hogy a C99 nyelv már több mint 15 éve létezik, a C fejlesztők körében nem terjedt el, és a legtöbb helyen ragaszkodnak a C89/C95 nyelvek alkalmazásához.

Néhány hasznos hivatkozás

A C99 szabvány
http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=29237

A C99 szabvány 3. technikai javítása (2007. szeptember 7.)
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

A C11 szabvány
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853/

A C1X szabvány utolsó munkaverziója (2011. április 12.)
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

C referencia
http://en.cppreference.com/w/c/language

C11 Wikipedia
http://en.wikipedia.org/wiki/C11_(C_standard_revision)

1 Változások a C nyelv alapelemei szintjén

1.1 Azonosítók hossza

A C99 lényegesen megnövelte az azonosítók nevének méretét. Az új szabvány szerint a fordító ***63*** karaktert vesz figyelembe a belső (**static**, **auto**, **register**) azonosítók és makrók nevéből, illetve ***31*** karaktert a külső (**extern**) azonosítók esetén. (Ezek a korlátok korábban *31* és *6* voltak.) Ugyancsak megváltozott a logikai forrássor maximális hossza, nevezetesen *509* karakterről ***4095*** karakterre.

1.2 Kulcsszavak

A C99 az alábbi új kulcsszavakat vezette be: **inline**, **restrict**, **_Bool**, **_Complex** és **_Imaginary**. Az utóbbi három a szokásos megoldásoktól eltérően aláhúzásjellel és nagybetűvel kezdődik. Ennek oka, hogy elkerüljék az ütközést a szokásos felhasználói nevekkel (*bool*, *complex* stb.). Azonban az <*stdbool.h*> deklarációs állomány makróként definiálja a ***bool*** nevet, valamint a szokásos logikai konstansokat ***false***, ***true***. Hasonlóképpen a <*complex.h*> fájl beépítésével elérhető a ***complex*** név, valamint (a szabvány szerint opcionális) ***imaginary*** név.

Ajánlott megoldás mindkét említett állomány beépítése a programunkba. A későbbi példákban ezt feltételezzük, amikor a ***bool***, ***complex***, ***imaginary*** neveket használjuk.

1.3 Megjegyzések

A C99 nyelvben a megjegyzések programban történő elhelyezésére a /* ... */ jeleken kívül a // (két perjel) is használható. A // jel alkalmazása esetén a megjegyzést nem kell lezárni, hatása a sor végéig terjed.

```
/* Az alábbi részben definiáljuk
   a szükséges változókat       */
int a = 7;      /* segédváltozó */

// Az alábbi részben definiáljuk
// a szükséges változókat
int a = 7;     // segédváltozó
```

1.4 Kibővített karaktertámogatás

A C nyelvet eredetileg nem nemzetközi programozási nyelvnek tervezték, karakterkészlete is az USA szabványokra épült. A C program bizonyos karaktereinek bevitele a billentyűzetről gondot okozhat, ezért a szabvány alternatív megoldásokat tartalmaz.

1.4.1 Háromjeles (trigraph) szekvenciák (C90)

trigraph	*szimbólum*	*trigraph*	*szimbólum*	*trigraph*	*szimbólum*
??=	#	??(	[	??/	\
??)	]	??'	^	??<	{
??!	|	??>	}	??-	~

```
??=include <stdio.h>
??=define MERET 80

int main()
??<
   int q??(MERET??) = ??<1,2,3,4??>;
   printf("%d??/n", q??(3??));
??>
```

```
#include <stdio.h>
#define MERET 80

int main()
{
   int q[MERET] = {1,2,3,4};
   printf("%d\n", q[3]);
}
```

1.4.2 Kétjeles (digraph) szekvenciák

digraph	*szimbólum*	*digraph*	*szimbólum*	*digraph*	*szimbólum*
<:	[	:>	]	<%	{
%>	}	%:	#	%:%:	##

```
%:include <stdio.h>
%:define MERET 80

int main()
<%
   int q<:MERET:> = <%1,2,3,4%>;
   printf("%d\n", q<:3:>);
%>
```

1.4.3 A logikai operátorok alternatív neve az ISO646 szerint

Az <*iso646.h*> deklarációs állomány tartalmazza a logikai operátorok ISO646 szabvány szerinti neveit:

makró	*operátor*	*makró*	*operátor*	*makró*	*operátor*
and	&&	and_eq	&=	bitand	&
bitor	|	compl	~	not	!
not_eq	!=	or	||	or_eq	|=
xor	^	xor_eq	^=		

1.4.4 Univerzális karakternevek (Universal Character Names – UCN)

Az UCN lehetővé teszi, hogy a C forrásprogramban tetszőleges karaktert megadjunk, az alábbi formákban:

```
\u 4_hexadecimális_jegy
\U 8_hexadecimális_jegy
```

Az univerzális karakternévben nem szerepelhet *00A0*-nál kisebb érték, kivéve a *0024* ($), a *0040* (@) és *0060* (?) kódokat, valamint a *D800* és *DFFF* közé eső számokat. Az UCN karaktereket azonosítókban, karakteres konstansokban, sztringliterálokban egyaránt alkalmazhatjuk:

```
int \u20ac = 0;  // €

char msg[] = "UCS karakterek \u20ac es \U000020ac";
```

2 Típusok és deklarációk

2.1 Az implicit int típus megszűnése

A C89 szabvány a program bizonyos helyein (függvény, paraméterek) megengedte a típus elhagyását, és automatikusan az **int** típust használta a fordítás során. Bár ezt a gyakorlatot kevesen követték, néha előfordult, és nehezen kideríthető programhibához vezetett. A C99 szabvány igényli a típusok megadását, így az alábbi program már nem fordítható le hibátlanul:

```
#include <stdio.h>

volatile v;

sum(a,b)
{
   return a+b;
}

main()
{
   printf("%d\n", sum(12,23));
}
```

A fordítás során az alábbi figyelmeztető üzenetek keletkeznek:

```
prog.c:4: warning: type defaults to 'int' in declaration of 'v'
prog.c:7: warning: return type defaults to 'int'
prog.c: In function 'sum':
prog.c:7: warning: type of "a" defaults to "int"
prog.c:7: warning: type of "b" defaults to "int"
prog.c: At top level:
prog.c:12: warning: return type defaults to 'int'
```

2.2 Új típusok

A C99 szabvány a kor igényeinek megfelelően új típusokkal bővíti a nyelvet. (A felsorolásban megadott nevek a már említett <*stdbool.h*> és <*complex.h*> deklarációs állományok beépítésével jönnek létre):

Típus	***Leírás***
long long int	(legalább) 64-bites előjeles egész,
unsigned long long int	(legalább) 64-bites előjel nélküli egész,
bool	logikai adattípus, amely 0 és 1 értékek tárolására képes,
float complex, **double complex**, **long double complex**	a hagyományos lebegőpontos típusoknak megfelelő komplex típusok.
float imaginary, **double imaginary**, **long double imaginary**	a hagyományos lebegőpontos típusoknak megfelelő imaginárius típusok, amelyek megléte opcionális.

A **long long int** és az **unsigned long long int** típusok teljes támogatást kapnak a C99 nyelv oldaláról. Minden egészekre vonatkozó művelet elvégezhető velük, továbbá megfelelő nevű könyvtári függvények segítik a használatukat. Konstans értékekben az ***ll***, ***LL***, illetve az ***ull***, ***ULL*** utótaggal jelölhetjük őket. A ***ll*** formátum-módosító segítségével a ***printf***() és ***scanf***() függvények is kezelik ezeket a nagyméretű egészeket.

```
long long int x=123456789012345678LL;
long long z=0LL;
unsigned long long int y = x / 712;
printf("y   = %llu\n", y);
printf("z   = ");
scanf("%lld", &z);
printf("z^2 = %lld\n", z*z);
```

```
y   = 173394366590373
z   = 1234567
z^2 = 1524155677489
```

A komplex típusokhoz a C99 nyelv biztosítja a konstans komplex értékek (például, *7 + 9*I, 1 - 2*_Complex_I*) használatát, az alapvető aritmetikai műveleteket valamint egy sor matematikai függvényt, azonban az I/O formátumokról nem rendelkezik. A szabványos eszközök alkalmazását az alábbi program szemlélteti:

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

int main()
{
    double complex a=1+2*I, b=-7*I, c =9;
    double re,im;

    printf("a = %7.2f+%7.2fi\n", creal(a), cimag(a));
    printf("b = %7.2f+%7.2fi\n", creal(b), cimag(b));
    printf("c = %7.2f+%7.2fi\n", creal(c), cimag(c));
    c = a + b;
    printf("a + b  = %7.2f+%7.2fi\n", creal(c), cimag(c));
    c = a * b;
    printf("a * b  = %7.2f+%7.2fi\n", creal(c), cimag(c));
    c = csqrt(a);
    printf("sqrt(a)= %7.2f+%7.2fi\n", creal(c), cimag(c));
    printf("c = ");
    if (2 == scanf("%lg + %lgi", &re, &im)) {
        c = re + im*I;
        printf("c = %7.2f+%7.2fi\n", creal(c), cimag(c));
    }
}
```

```
a =    1.00+   2.00i
b =    0.00+  -7.00i
c =    9.00+   0.00i
a + b  =    1.00+  -5.00i
a * b  =   14.00+  -7.00i
sqrt(a)=    1.27+   0.79i
c = 12.23 + -7.29i
c =   12.23+  -7.29i
```

2.3 Kibővített egész típusok

A C99 szabvány megvalósításai további (8-, 16-, 32- vagy 64-bites) egész típusokat definiálhatnak, melyekhez tartozó neveket az <*stdint.h*> és <*inttypes.h*> deklarációs állományok hozzák létre. Ott a típusok mellett értékhatárokat és ***printf()***, ***scanf()*** formátumelemeket is találunk:

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

int main()
{
    int8_t a=INT8_MIN, b=INT8_MAX;
    printf("%6"PRId8",%6"PRId8"\n", a, b);
    printf("a=");
    scanf("%"SCNd8, &a);
    b = a / 7;
    printf("%6"PRId8"\n", a);
}
```

```
  -128,    127
a=123
   123
```

2.4 Lebegőpontos típusok

A C99 szabvány a **float**, **double** és a **long double** lebegőpontos típusok körét nem bővíti, azonban több területen új lehetőségeket kínál a programozók számára.

Hexadecimális formában a lebegőpontos konstans értékeket pontosabban adhatjuk meg (a mantisszát 16-os, míg a ***p/P*** betű után a 2 kitevőjét 10-es számrendszerben kell szerepeltetnünk.) Például, a *0x1.004p7* decimális értéke *128.125*. Az ilyen adatok kezelését segítik az új **%a** és **%A** *printf/scanf* konverziós formátumok.

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

int main()
{
    printf("%8.3f\n",  0x1.ap+0f);
    printf("%8.3f\n",  0x1.0P+10);
    printf("%17.15f\n", 0x0.C90FDAA22168CP2);

    printf("%A\n", 7.29);
    printf("%a\n", 2.0/3.0);

    printf("1/0.0 = %f\n", 1/0.0);
    printf("-1/0.0 = %f\n", -1/0.0);
    printf("0.0/0.0 = %f\n", 0.0/0.0);
    printf("sqrt(-1.0) = %f\n", sqrt(-1.0));
    printf("INFINITY/INFINITY = %f\n", INFINITY/INFINITY);
}
```

```
   1.625
1024.000
3.141592653589793
0X1.D28F5C28F5C29P+2
0x1.5555555555555p-1
1/0.0 = inf
-1/0.0 = -inf
0.0/0.0 = nan
sqrt(-1.0) = nan
INFINITY/INFINITY = nan
```

A C99 egy opcionális specifikációt biztosít az ***IEEE*** lebegőpontos aritmetika működésének szabályozására: a végtelen, a *NAN* (nem szám) és az előjeles nullák kezelésének, a kifejezések kiértékelésének valamint a konverziós módok beállítására. (*math.h*, *fpenv.h*)

A kibővített matematikai függvények a három lebegőpontos típuson (például ***sqrt()***, ***sqrtf()***, ***sqrtl()***) túlmenően a komplex típusokra (például ***csqrt()***, ***csqrtf()***, ***csqrtl()***) is alkalmazhatók. (*math.h*, *complex.h*). A megfelelő függvény kiválasztásában segítenek a <*tgmath.h*> deklarációs állomány makrói (például ***sqrt()***).

2.5 Változó méretű tömbök

A C89-ben a tömbök a fordítás során jönnek létre, a méretet definiáló konstans kifejezések felhasználásával. A C99 a változóméretű tömbök (*variable-length array – VLA*) bevezetésével bővíti a tömbök használatának lehetőségeit. A futásidőben létrejövő VLA csak automatikus élettartamú, lokális (**auto**, **register**) változó lehet, amely a függvénybe való belépéskor jön lére, és a függvényből való kilépéskor automatikusan megszűnik. A tömb méretét tetszőleges egész típusú kifejezéssel megadhatjuk, azonban a létrehozást követően a mérete nem változtatható meg.

A változó méretű tömbökkel a **sizeof** operátor futásidejű változatát alkalmazza a fordító, ahogy az a példaprogramban jól nyomonkövethető.

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

void f2dimVLA(int m, int n)
{
    double matrix[m][n];
    printf("%d\n", sizeof(matrix));  // mérete m*n*8
}

void funcVLA(int n, int p[n])
{
    // p egy VLA-ra mutató pointer, mérete 4
    printf("%d\n", sizeof(p));
    for (int i=0; i<n; i++)
       p[i] = i*i;
}

int main()
{
    int meret = 10;
    int aVLA[meret];
    printf("%d\n", sizeof(aVLA)); // a tömb mérete 10*4
    funcVLA(meret, aVLA);
    f2dimVLA(7,5);
}
```

2.6 A restrict típusminősítő

A C99 szabvány által bevezetett megoldások közül az egyik leglényegesebb újdonság a **restrict** (korlátozott) típusminősítő. Ezt a típusminősítőt csak mutatókhoz használhatjuk. A minősítő azt jelzi a fordítónak, hogy a mutatott objektumhoz csak az adott mutatóval fér hozzá a program.

Például, ha egy függvény két **restrict** mutató paraméterrel rendelkezik, akkor ez informálja a fordítót, hogy a két mutató különböző (egymást nem átfedő) objektumokra mutatnak, így hatékonyabb futtatható kód jöhet létre.

A hatékonyabb kódgenerálás mellett, a C szabványos könyvtár bizonyos függvényeinek újradeklarálásával, a programozó is pontosabb információkhoz jut. Például C89-ben a ***memcpy()*** függvény prototípusából nem derül ki, hogy a függvény hibásan működhet egymást átfedő memóriablokkok esetén:

```
void *memcpy(void *dest, const void *src, size_t n);
```

C99-ben jól látható, hogy a függvény csak egymást nem átfedő objektumok esetén alkalmazható:

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
```

Ezzel szemben a ***memove()*** minden esetben jól (de lassabban) működik:

```
void *memmove(void * s1, const void * s2, size_t n);
```

Az alábbi példaprogram a **restrict** típusminősítő használatát szemlélteti:

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

float *vektor(size_t n, float v) {
   float *p = malloc(n * sizeof(float));
   assert(p);
   for (size_t i = 0; i < n; i++)
     p[i] = v;
   return p;
}
```

```
void novel(int n, float * restrict a1, const float * restrict a2) {
   for (int i = 0; i < n; i++)
      a1[i] += a2[i];
}

int main(void) {
   const int n = 10;
   float * restrict a = vektor(n,12),
         * restrict b = vektor(n,23);
   novel(n, a, b);
}
```

2.7 Ismétlődő típusminősítők

Amennyiben egy típusminősítő (**const**, **volatile,** **restrict**) többször előfordul a típusfelírásban, az azonos minősítők közül csak az elsőt veszi figyelembe a fordító. Az ismétlődés közvelenül megadva, vagy közvetetten – a **typedef** alkalmazásával – egyaránt előfordulhat:

```
const volatile const int bd = 729;
```

ugynaz, mint a

```
const volatile int bd = 729;
```

3 Programstruktúrák

3.1 Fordítói korlátozások módosítása

Az alábbi táblázatban összefoglaltuk az ide vonatkozó értékeket C89 és C99 nyelvek esetén:

Korlátozás	*C89*	*C99*
Blokkok egymásba ágyazásának maximális mélysége	*15*	*127*
Feltételes beépítés egymásba ágyazásának maximális mélysége	*8*	*63*
Struktúra és unió tagjainak maximális száma	*127*	*1023*
Függvényargumentumok maximális száma	*31*	*127*

3.2 Deklarációk és definíciók bárhol megadhatók a blokkon belül

A C99 szabvány szerint az utasításblokkon belül az utasításokat és a definíciókat/deklarációkat tetszőleges sorrendben megadhatjuk. Egyetlen feltétel, hogy a változót mindenképpen deklarálnunk/definiálnunk kell a felhasználása előtt. Élve ezzel a lehetőséggel a változók definíciója és a felhasználása közel helyezhető, elkerülve ezzel bizonyos programozási hibákat:

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

int main()
{
    const double fi = (sqrt(5)+1)/2; // aranyarány
    double a;
    printf("a = ");
    scanf("%lf", &a);
    double b = a / fi;
    printf("b = %5.2f\n", b);  // aranymetszés szerint
}
```

```
a = 729
b = 450.55
```

3.3 Változók definíciója a for cikluson belül

A **for** ciklusfej inicializációs részében változókat is definiálhatunk, melyek élettartama a ciklusra korlátozódik:

```
#include <stdio.h>

int main()
{
    int  m = 80;
    for (int m=10, n=20; m < n; m++, n--)
    {
       printf("%4d", m);
    }
    printf("\n%4d\n", m);
}
```

```
  10  11  12  13  14
  80
```

3.4 Minősített kezdőértékek struct, union és tömbtípusok esetén

Struktúrák és unionok esetén a tagnévvel, tömbök esetén pedig az indexeléssel meghatározhatjuk, hogy az inicializáció melyik tagra/elemre vonatkozik. A felhasználói típusok esetén a minősítő név előtt a pontot kell szerepeltetnünk. Nézzünk néhány esetet!

```
#include <stdio.h>

    struct st1 {
       int a,b;
       char n[10];
    };
```

```
    struct st2 {
       struct st1 s;
       int c;
    };

int main()
{
   int a[7] = {[4]=12, 23,  [0]=7, 29};

   struct st1 s1 = {.b=12, .n="C99" };                    //a=0
   struct st1 s2 = {.a=23, .n[0]='C', .n[1]='#', .n[2]=0}; //b=0
   struct st2 s3 = {.c = 7, .s.a = 29, .s.n="C++"};       //s.b=0
   struct st2 s4 = {.s.a =1, 2, "NO", 3 };
}
```

Amennyiben a minősített és a hagyományos inicializálókat keverjük, a minősített után a definíció sorrendjében kapnak értéket az elemek/tagok. Emiatt a fenti példában szereplő *a* tömb a következő-képpen inicializálódik:

0.	1.	2.	3.	4.	5.	6.
7	29	0	0	12	23	0

3.5 Összetett típusú literálok (értékek)

Nagyban egyszerűsítheti a stuktúrákat, uniókat és tömböket használó C99 programokat, hogy az ilyen típusú változóknak nemcsak a definicíó során, hanem bármikor adhatunk értéket. Az összetett típusú literál hagyományos, vagy minősített kezdőértéklistából és egy típusátalaktásból áll:

(*összetett típus*) { *értéklista* }

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

int main()
{
   struct  vektor {int a,b;};
   const double pi = 3.14159265;

   double *pv, d;
   d = 3;
   pv = (double[]) {2*sin(pi/d), 3*cos(pi/(d+1))};

   const char * ccp = (const char *) {"C nyelv"};

   struct vektor x, y;
   x = (struct vektor) {.b = 7, .a = 8};
   y = (struct vektor) {2, 9};

   int* m[] = { (int []) {1},
                (int []) {2,3},
                (int []) {4,5,6} };
   for (int i=0; i<sizeof(m) / sizeof(int*); i++)
   {
       for (int j=0; j<=i; j++)
           printf("%5d\t", m[i][j]);
       printf("\n");
   }
}
```

3.6 Rugalmas méretű tömb struktúratag

A C99 lehetővé teszi, hogy egy struktúra utolsó tagjaként méret nélküli egydimenziós tömböt adjunk meg. A tömb akkor jön létre, amikor a stuktúra és a tömb számára dinamikusan helyet foglalunk a memóriában. Fontos megjegyeznünk, hogy a struktúrában legalább egy szokásos tagnak meg kell előznie a rugalmas tömbtagot.

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct PASstring {            // Pascal sztring
    unsigned short hossz;
    char s[];                 // rugalmas tömbtag
};
// C sztring tárolása Pascal sztringben
struct PASstring *CtoPAS(const char *cstr)
{
    struct PASstring *pas;
    size_t h = strlen(cstr);
    // helyfoglalás a PASstring stuktúra és az s tömb számára
    pas = malloc(sizeof (struct PASstring) + h);
    assert(pas != NULL);
    pas->hossz = h;
    // karakterek másolása a záró 0-ás bájt nélkül
    memcpy(pas->s, cstr, h);
    return pas;
}

int main()
{
    char cstring[] = "A C programozasi nyelv";
    struct PASstring *pstring = CtoPAS(cstring);
    // adott méretű sztring megjelenítése (*s)
    printf("hossz=%hu, s=\"%.*s\"\n", pstring->hossz,
                          pstring->hossz, pstring->s);
    free(pstring);
    return EXIT_SUCCESS;
}
```

3.7 Változások az egész típusok közötti konverziókban

C89-ben a **char**, **short int** és az **int** típusú bitmezőket a kifejezésekben **int** vagy **unsigned int** helyén használhattuk. Amennyiben egy konvertált érték belefért az **int** típusba, akkor az átalakítás **int** típusra történt, ellenkező esetben pedig **unsigned int** típusra.

C99-ben minden egész típushoz tartozik egy rang. Például, a **long long** típus rangja magasabb, mint az **int** típus rangja, ami magasabb a **char** típusénál. A kifejezésekben minden olyan egész típus, amelynek rangja alacsonyabb, mint az **int** vagy az **unsigned int** rangja, felhasználható az **int**, illetve az **unsigned int** helyett.

4 Függvények

4.1 Nincs implicit függvénydeklaráció

Amikor a C89 szabvány szerinti fordító egy nem deklarált függvényre való hivatkozást talál, akkor az implicit függvénydeklaráció alapján végezi el a fordítást, melynek formája:

```
extern int függvénynév();
```

A C99 **nem támogatja** az implicit függvénydeklaráció alkalmazását.

4.2 A return szigorítása

A C89 szabvány megengedi, hogy a **void** típustól eltérő típusú függvényből érték nélküli **return** utasítással lépjünk ki, mikor is a függvényérték definiálatlan:

```
int fv(void)
{
   // ...
   return ;
}
```

C99 szerint a fenti esetekben kilépési érték megadása **kötelező**, az üres **return** csak **void** típusú függvényekben használható.

4.3 Inline függvények

Az **inline** („soron belüli”) függvényekkel elsősorban a paraméterezett ***#define*** makrókat helyettesíthetjük. Az **inline** függvények használatának előnye, hogy a függvényhíváskor az argumentumok feldolgozása teljes körű típusellenőrzés mellett megy végbe. Az **inline** esetén a fordító optimalizálja a függvényhívást, ami általában a függvény törzsét képező kód behelyettesítését jelenti a függvényhívás helyére („kódmakró”). Például, a *MAX* makró helyett egész értékek esetén *a max()* **inline** függvény használata javasolt.

```
#define MAX(x,y) (x)>(y)?(x):(y)

inline int max(int x, int y)
{
   return (x>y?x:y);
}
```

Nézzük a hivatkozást a makróra és az **inline** függvényre!

```
int main()
{
   int a;
   a=MAX(4,29); // az előfordító a=(4)>(29)?(4):(29); utasítássá
                // alakítja át.

   a=max(4,29); // a fordító a függvény törzsét képező kódot
                // helyettesíti be az utasításba: a=x>y?x:y;
}
```

Az **inline** definíció csak javaslat a fordító számára, amelyet az bizonyos feltételek esetén (de nem mindig!) figyelembe is vesz. Általában kisméretű, gyakran hívott függvények esetén ajánlott alkalmazni ezt a megoldást.

4.4 Típusmódosítók a tömbparaméterek deklarációjában

A C99 szabvány megengedi, hogy a tömbparaméterekben, a szögletes zárójelek között a típusmódosítókat és a **static** kulcsszót használjuk.

Tömbparaméter	*Pointeres megfelelő*
`void fv1( int a[100]);`	`void fv1(int *a);`
`void fv2( int a[const 100]);`	`void fv2(int *const a);`
`void fv3( int a[volatile 100]);`	`void fv3(int *volatile a);`
`void fv4( int a[restrict 100]);`	`void fv4(int *restrict a);`
`void fv5( int a[static 100]);`	–
`void fv6( int a[static const 100]);`	–

A **const**, **volatile** és **restrict** típusminősítők bevitelével a tömbparaméter szögletes zárójelei közé, magára a tömbre mutató pointerre adhatunk meg előírásokat (konstans, aszinkron módon megváltozhat, egyetlen hivatkozás a tömbre). Ilyenkor a megadott méretadattal nem foglalkozik a fordító, ami azonnal érhető a pointeres megfelelőkből.

Egészen más a helyzet a **static** kulcsszó alkalmazásával, mely után a méretmegadás kötelező. Ezzel közöljük a fordítónak, hogy az átadott mutató biztosan nem ***NULL***, és a tömb legalább a megadott számú elemet tartalmazza. Az információk birtokában a fordító hatékonyabb futtatható kódot képes létrehozni. Felhívjuk a figyelmet ara, hogy a prototípusban előírt méret meglétéről a függvény hívójának kell gondoskodnia, mivel a fordító nem tudja azt ellenőrizni. A megadottnál kisebb méretű tömbbel a függvény valószínűleg hibásan fog működni.

4.5 A __func__ előre definiált azonosító

A C99 fordító minden függvényben definiál egy (**static const char** tömb típusú) ***__func__*** azonosítót, amely az adott függvény nevét tárolja. Ez jól használható nyomkövetés, hibakeresés során.

```
#include <stdio.h>

void fuggveny()
{
   printf("%s\n", __func__); // fuggveny
}

int main()
{
   printf("%s\n", __func__); // main
   fuggveny();
   return 0;
}
```

4.6 Új deklarációs állományok – új könyvtári elemek

A C99 egy sor új könyvtári elemmel bővíti a C nyelvet, és sok jól ismert függvényt is újradefiniál. Az alábbiakban az új deklarációs állományokat szedtük csokorba.

Fejléc	*Leírás*
<*complex.h*>	Az új komplex típusokkal való műveletvégzés függvényei.
<*fenv.h*>	A lebegőpontos környezet kezelését segítő függvények és makrók.
<*inttypes.h*>	Beépíti az *stdint.h* fájlt, makrókat definiál a ***scanf()*** és ***printf()*** függvények használatához, valamint függvényeket az ***intmax_t*** típushoz.
<*iso646.h*>	A logikai műveletek alternatív nevei (1995-től).
<*stdbool.h*>	Makrók (***bool***, ***false***, ***true***) az új logikai típus használatához.
<*stdint.h*>	Rögzített méretű egész típusok és a hozzájuk tartozó makrók.
<*tgmath.h*>	A szokásos matematikai műveleteket lefedő, általánosított típusú paraméterekkel rendelkező (*type-generic*) makrók.
<*wchar.h*>	Széles karakterek osztályozása és átalakítása: ***isw****…()*, ***tow****…()* (1995-től).
<*wctype.h*>	Széles karakterekből álló karaktersorozatok és memóriapufferek kezelése: ***wcs****…()*, ***memw****…()* (1995-től).

5 Előfordító

5.1 Új előre definiált szimbolikus konstansok

A C99 szabvány az alábbi makrók definiálását javasolja a fordítónak:

__STDC_HOSTED__	1, ha a futtatói környezet adottságai megfelelnek a C szabvány előírásainak.
__STDC_VERSION__	Értéke *199409L* a C95, míg *199901L* a C99 szabványú fordító esetén.
__STDC_IEC_559__	1, ha az IEC 60559 lebegőpontos aritmetikát támogatja a fordító.
__STDC_IEC_599_COMPLEX__	1, ha az IEC 60559 komplex aritmetikát támogatja a fordító.
__STDC_ISO_10646__	A fordító által támogatott ISO/IEC 10646 specifikáció dátuma *yyyymmL* formában.

5.2 Beépített fordítási előírások (pragmák)

A C99 szabvány az alábbi pragmákat definiálja:

STDC FP_CONTRACT ON/OFF/DEFAULT
: Bekapcsolva: a lebegőpontos kifejezéseket a hardveralapú eljárások oszthatatlan egységekként kezelik. Az alapértelmezett állapot realizáció-függő.

STDC FENV_ACCESS ON/OFF/DEFAULT
: Jelzi a fordítónak, hogy a lebegőpontos környezet elérhető. Az alapértelmezett állapot realizáció-függő.

STDC CX_LIMITED_RANGE ON/OFF/DEFAULT
: Jelzi a fordítónak, hogy bizonyos, összetett értékű kifejezések biztonságosak. Alapértelmezett állapot az OFF.

5.3 A _Pragma operátor

A ***_Pragma*** operátor segítségével a szokásos ***#pragma*** direktíva mellett dinamikusan is definiálhatunk fordítási előírásokat. Például a

```
_Pragma ( "align(power)" )
```

megfelel a

```
#pragma align(power)
```

direktívának.

5.4 Makrók változó-hosszúságú paraméterlistával

A három pont makrók esetén is használható, és a megadott argumentumokat a makrón belül a ***__VA_ARGS__*** szimbólum tartalmazza:

```
#define trace(...) fprintf(stderr, __VA_ARGS__)

#define eprintf(format, ...) fprintf(stderr, format, ##__VA_ARGS__)

#define textout(...) puts(#__VA_ARGS__)
```

A fenti makrókat hívó programrészlet:

```
textout(Variadic makro peldak);
trace("start\n");
trace("%d\n", 123);
eprintf("%d %s\n", 123, __func__);
eprintf("vege\n");
```

A programrészlet az előfordítás után:

```
puts("Variadic makro peldak");
fprintf(stderr, "start\n");
fprintf(stderr, "%d\n", 123);
fprintf(stderr, "%d %s\n", 123, __func__);
fprintf(stderr, "vege\n");
```

6. A szabványos C99 nyelv könyvtárának áttekintése

6.1 A szabványos C nyelv deklarációs állományai

A teljesség kedvéért a **C99** szabvány új állományait is szerepeltetjük a felsorolásban. A *C95* névvel a C89 szabvány 1995-ben született normatív kiegészítésére (*Normative Addendum 1*) hivatkozunk.

Deklarációs állomány	***C szabványtól létezik***
<assert.h>	
<complex.h>	C99
<ctype.h>	
<errno.h>	
<fenv.h>	C99
<float.h>	
<inttypes.h>	C99
<iso646.h>	C95
<limits.h>	
<locale.h>	
<math.h>	
<setjmp.h>	

Deklarációs állomány	***C szabványtól létezik***
<signal.h>	
<stdarg.h>	
<stdbool.h>	C99
<stddef.h>	
<stdint.h>	C99
<stdio.h>	
<stdlib.h>	
<string.h>	
<tgmath.h>	C99
<time.h>	
<wchar.h>	C95
<wctype.h>	C95

Felhívjuk a figyelmet, hogy az újabb szabványok megjelenésekor a régebbi deklarációs állományok tartalma módosulhat, bővülhet, sőt bizonyos függvények működése/paraméterezése is megváltozhat.

Az alábbiakban a felhasználás célja szerint csoportosítjuk a deklarációs állományokat. A táblázatban egy csillaggal jelöltük a C szabvány 1995-ös módosításával (*C95*) megjelent fájlokat, két csillaggal pedig a C99 szabvánnyal bevezette állományokat.

A C nyelv támogatása	
Típusok és makrók (***NULL***, ***size_t*** stb.).	<stddef.h>
Makrók (***bool***, ***false***, ***true***) az új logikai típus használatához.	<stdbool.h>**
Beépíti az *stdint.h* fájlt, makrókat definiál a ***scanf()*** és ***printf()*** függvények használatához, valamint függvényeket az ***intmax_t*** típushoz.	<inttypes.h>**
Rögzített méretű egész típusok és a hozzájuk tartozó makrók.	<stdint.h>**
Dinamikus memóriakezelés.	<stdlib.h>
A változó hosszúságú argumentumok kezelését segítő makrók (***va_***...).	<stdarg.h>
Programindítás és -befejezés.	<stdlib.h>
A C nyelv hagyományos matematikai függvényei.	<math.h>
Az új komplex típusokkal való műveletvégzés függvényei.	<complex.h>**
A szokásos matematikai műveleteket lefedő, általánosított típusú paraméterekkel rendelkező (*type-generic*) makrók.	<tgmath.h>**
Az implementációt jellemző típus-értékhatárok (***_MIN***, ***_MAX***, ***_EPSILON***, ***_DIG***).	<float.h> <limits.h>
A lebegőpontos környezet kezelését segítő függvények és makrók.	<fenv.h>**
A logikai műveletek alternatív nevei.	<iso646.h>*

Általános szolgáltatások	
Szabványos adatbevitel és adatkivitel.	<stdio.h>
Dátum- és időkezelés.	<time.h>
Karakterek osztályozása és átalakítása (***is…()***, ***to…()***).	<ctype.h>
Széles karakterek osztályozása és átalakítása (***isw…()***, ***tow…()***).	<wctype.h>*
Karaktersorozatok és memóriapufferek kezelése (***str…()***, ***mem…()***).	<string.h>

Széles karakterekből álló karaktersorozatok és memóriapufferek kezelése (***wcs**…()*, ***memw**…()*).	`<wchar.h>*`
Sztringet számmá átalakító függvények (***ato**…()*, ***strto**…()*).	`<stdlib.h>`
A helyi (országfüggő) beállítások kezelését segítő típusok és függvények.	`<locale.h>`
Speciális algoritmusok (rendezés, véletlenszám előállítása stb.)	`<stdlib.h>`

Hibakezelés

Az ***assert()*** makró.	`<assert.h>`
Általános hibatároló (***EDOM***, ***ERANGE***, ***errno***).	`<errno.h>`
A program futása során jelzések (szignálok) küldését és feldolgozását segítő könyvtár.	`<signal.h>`
Függvények közötti ugrásokat támogató típus és függvények.	`<setjmp.h>`

6.2 A szabványos C99 nyelv új könyvtári függvényei és makrói

Megjegyezzük, hogy a C99 szabvány könyvtárában a legtöbb mutató típusú paraméterek mellett szerepel a **restrict** típusmódosító, amely lehetőséget biztosít hatékonyabb kód előállítására.

6.2.1 A C nyelv támogatása

Általános típus- és makródefiníciók

deklarációs állomány: *stddef.h*

típusok és makrók: ***wchar_t***

A logikai típushoz tartozó szabványos nevek

deklarációs állomány: *stdbool.h*

típusok és makrók ***bool***, ***true***, ***false***, ***__bool_true_false_are_defined***

6.2.2 Változó hosszúságú argumentumlista kezelése

Deklarációs állomány: *stdarg.h*

Makró: ***va_list***

Makró	*Leírás*
va_copy()	A ***va_list*** tartalmának másolása.

6.2.3 Karakterek osztályozása és konverziója

Egybájtos karakterek esetén a deklarációs állomány: *ctype.h*

Függvény	*Leírás*
isblank()	Szavak tagolására használt karakter.

Széles karakterek esetén a deklarációs állomány: *wctype.h*

Típusok és konstansok: ***wint_t***, ***wctrans_t***, ***wctype_t***, ***WEOF***

Függvény	*Leírás*
iswalnum()	Alfanumerikus széles karakter tesztelése.
iswalpha()	Alfabetikus széles karakter tesztelése.
iswblank()	Szavak tagolására használt széles karakter.
iswcntrl()	Széles vezérlőkarakter tesztelése.
iswdigit()	Decimális számjegy tesztelése.
iswgraph()	Grafikus széles karakter tesztelése (a szóköz nem).
iswlower()	Angol kisbetű tesztelése.
iswprint()	Kinyomtatható széles karakter tesztelése (a szóköz is).
iswpunct()	Írásjel széles karakter tesztelése.
iswspace()	Szóköz, tabulátor, soremelés vagy lapdobás széles karakter tesztelése.
iswupper()	Angol nagybetű tesztelése.
iswxdigit()	Hexadecimális számjegy tesztelése.
iswctype()	Széles karakter karakterosztályhoz való tartozásának ellenőrzése.
wctype()	Az ***iswctype()*** függvény hívásakor használható karakterosztállyal tér vissza.
tolower()	Széles karakter tesztelése és átalakítása kisbetűvé, ha az nagybetű.
toupper()	Széles karakter tesztelése és konvertálása nagybetűvé, ha az kisbetű.
towctrans()	Széles karakter kódolása táblázat alapján.
wctrans()	A ***towctrans()*** függvény használatához szükséges kódtáblával tér vissza.

6.2.4 Sztring tartalmának számmá alakítása

Deklarációs állomány: *stdlib.h*

Függvény	*Leírás*
atoll()	Sztring átalakítása **long long int** típusú értékké.
strtof()	Sztring konvertálása **float** típusú értékké.
strtold()	Sztring átalakítása **long double** típusú értékké.
strtoll()	Sztring átalakítása **long long int** típusú értékké.
strtoull()	Sztring konvertálása **unsigned long long int** típusú értékké.

Széles karakterek esetén a deklarációs állomány: *wchar.h*

Típusok és konstansok: ***wchar_t***, ***size_t***, ***wint_t***, ***NULL***, ***WCHAR_MAX***, ***WCHAR_MIN***

Függvény	*Leírás*
wcstod()	Széles karakteres sztring konvertálása **double** típusú számmá.
wcstof()	Széles karakteres sztring konvertálása **float** típusú értékké.
wcstold()	Széles karakteres sztring átalakítása **long double** típusú értékké.
wcstol()	Széles karakteres sztring konvertálása **long int** típusú számmá.
wcstoll()	Széles karakteres sztring átalakítása **long long int** típusú értékké.
wcstoul()	Széles karakteres sztring konvertálása **unsigned long int** típusú számmá.
wcstoull()	Széles karakteres sztring konvertálása **unsigned long long int** típusú értékké.

6.2.5 Sztringkezelés

Széles karakterekből álló sztringek kezelésére a C99 szabvány a hagyományos (**char**) karaktersorozat-kezelő függvények széles karakteres változatait is tartalmazza

Deklarációs állomány: *wchar.h*

Típusok és konstansok: ***wchar_t***, ***size_t***, ***wint_t***, ***NULL***, ***WCHAR_MAX***, ***WCHAR_MIN***

Függvény	*Leírás*
wcscat()	Sztringek összekapcsolása.
wcschr()	Karakter első előfordulásának keresése sztringben.
wcscmp()	Két sztring összehasonlítása.
wcscoll()	A helyi (setlocale()) beállítások alapján hasonlít össze két sztringet.
wcscpy()	Sztring átmásolása egy másikba. A függvény az eredménysztring elejére mutató pointert ad vissza.
wcscspn()	Egy karakterkészlet sztring karaktereinek keresése másik sztringben.
wcslen()	A sztring hosszának lekérdezése.
wcsncat()	Valamely sztring adott számú karakterének hozzáfűzése egy másik sztringhez.
wcsncmp()	Két sztring adott számú karakterének összehasonlítása.
wcsncpy()	Valamely sztringből adott számú karakter átmásolása egy másik sztringbe.
wcspbrk()	Valamely sztring karaktereinek keresése egy másik sztringben.
wcsrchr()	Adott karakter utolsó előfordulásának keresése sztringben.
wcsspn()	Valamely sztring első olyan részsztringjének megkeresése, amelynek karakterei egy másik sztringben adottak.
wcsstr()	Valamely sztring első előfordulásának megkeresése egy másik sztringben.
wcstok()	Sztring részekre bontása. Az elválasztó karakterek egy másik sztringben adottak.
wcsxfrm()	A második argumentumban szereplő sztringet átalakítja és az első argumentum által kijelölt helyre másolja.

6.2.6 Memóriapufferek használata

Széles karakterek esetén a deklarációs állomány: *wchar.h*

A *string.h* állományban deklarált függvények **void *** pointerrel azonosítják a puffereket, míg ***wchar.h*** elemei a **wchar_t *** típust használják.

Típusok és konstansok: ***wchar_t***, ***size_t***, ***wint_t***, ***NULL***, ***WCHAR_MAX***, ***WCHAR_MIN***

Függvény	*Leírás*
wmemchr()	Egy mutatót ad vissza, amely egy adott karakter első előfordulását jelzi a pufferben.
wmemcmp()	Két pufferen belül adott számú karakter összehasonlítása.
wmemcpy()	Adott számú karakter másolása egyik pufferből a másikba.
wmemmove()	Adott számú karakter mozgatása egyik pufferből a másikba. (Egymást átfedő pufferek esetén is jól működik.)
wmemset()	A pufferben adott számú bájt feltöltése adott karakterrel.

6.2.7 Valós matematikai műveletek

A C nyelvre jellemző sokszínűség a matematikai függvények csoportját is áthatja. A szokásos függvényeken (*abs*, *sin*, *cos*, *sqrt*) túlmenően egy sor különleges függvényt találunk bene.

Deklarációs állomány: *math.h*

A C99 szabvány definiálja továbbá a hagyományos **double** típusú matematikai függvények **float** és **long double** megfelelőit, a név mögött az *f* illetve *l* betű megadásával. Például:

```
double sin(double);
float sinf(float);
long double sinl(long double);
```

A C99 ki is bővíti a matematikai függvények sorát, és biztosítja mindhárom valós típushoz a megfelelő változatot. (Az alábbi táblázatban csak a **double** típusú függvények nevei szerepelnek.)

Függvény	*Leírás*
acosh()	Arkusz koszinusz hiperbolikus függvény.
asinhf()	Arkusz szinusz hiperbolikus függvény.
atanh()	Arkusz tangens hiperbolikus függvény.
cbrt()	Köbgyökfüggvény.
copysign(x,y)	A visszaadott érték x abszolút értéke, előjele pedig megegyezik y előjelével.
erf()	Hibafüggvény.
erfc(()	Kiegészítő hibafüggvény.
exp2()	2 hatványa ($y=2^x$)
expm1(x)	e^x-1 függvény értéke.
fdim(x,y)	A két argumentum pozitív különbségének számítása (=*fmax(x-y,0)*)
fma(x,y,z)	x*y+z függvény értéke.
fmax()	A két argumentum közül a nagyobb egyenlő értékkel tér vissza.
fmin()	A két argumentum közül a kisebb egyenlő értékkel tér vissza.
hypot(x,y)	Derékszögű háromszög befogóihoz tartozó átfogó kiszámítása.
ilogb()	A kettes számrendszerben ábrázolt lebegőpontos szám kitevője egészként.
lgamma()	A gamma-függvény abszolút értékének természetes alapú logaritmusa.
llrint()	Kerekítés a legközelebbi egészre (**long long** típussal tér vissza).
lrint()	Kerekítés a legközelebbi egészre (**long** típussal tér vissza).
llround()	Kerekítés a nulla felőli legközelebbi egészre (**long long** típussal tér vissza).
lround()	Kerekítés a nulla felőli legközelebbi egészre (**long** típussal tér vissza).
log1p(x)	1+x természetes alapú logaritmusa.
log2()	Kettesalapú logaritmus.
logb()	A kettes számrendszerben ábrázolt lebegőpontos szám kitevője valósként.
nan(s)	A *NAN* (nem szám) értékkel tér vissza.
nearbyint()	Az argumentum kerekítése egész értékre.
nextafter(x,y)	Megadja a következő ábrázolható értéket (*y* irányában).
nexttoward(x,y)	Mint a ***nextafter()***, azonban y **long double** típusú
remainder(x,y)	Kiszámolja az osztási maradékot.
remquo(x,y,p)	Mint a ***remainder()***, azonban a hányadost is visszaadja (*p* mutató).
rint()	Egészre kerekít, azonban hibát hoz létre, ha az eredmény különbözik az argumentumtól.
round()	Kerekítés **int** típusú egészre.
scalbln(x,n)	x * ***FLT_RADIX***^{n} (*n* **long** típusú)
scalbn(x,n)	x * ***FLT_RADIX***^{n} (*n* **int** típusú)
tgamma()	Gamma-függvény.
trunc()	Kerekítés a legközelebbi egészre, nulla irányában.
fpclassify()	A makró minősíti az argumentum értékét (*NaN*, normalizált, végtelen stb.)
isfinite()	Nem nulla értékkel tér vissza, ha az argumentum véges.
isgreater(x,y)	x > y?
isgreaterequal(x,y)	x ≥ y?
isinf()	Nem nulla értékkel tér vissza, ha az argumentum végtelen.
isless(x,y)	x < y?
islessequal(x,y)	x ≤ y?
islessgreater(x,y)	x < y vagy x > y?
isnan()	Nem nulla értékkel tér vissza, ha az argumentum nem szám (*NaN*).
isnormal()	Nem nulla értékkel tér vissza, ha az argumentum normalizált.
isunordered(x,y)	Nem összehasonlítható?
signbit()	Nem nulla értékkel tér vissza, ha az argumentum negatív.

Deklarációs állomány: *stdlib.h*

Típusok és konstansok: ***lldiv_t***

Függvény	*Leírás*
llabs()	Egész abszolút érték (**long long**).
lldiv()	Egész osztás (**long long**).

6.2.8 Komplex matematikai műveletek

Deklarációs állomány: *complex.h*

Típusok és konstansok: ***complex***, ***_Complex_I***, ***imaginary***, ***_Imaginary_I***, ***I***

A táblázat soraiban az első függvény **double** illetve **double _Complex**, az *f* végűek **float**/**float _Complex**, míg az *l* végűek **long double**/**long double _Complex** típusokkal működnek.

Függvény	*Leírás*
cabs(), cabsf(), cabsl()	Komplex szám abszolút értéke.
cacos(), cacosf(), cacosl()	Komplex arkusz koszinusz.
cacosh(), cacoshf(), cacoshl()	Komplex arkusz koszinusz hiperbolikus.
carg(), cargf(), cargl()	Komplex szám argumentuma.
casin(), casinf(), casinl()	Komplex arkusz szinusz.
casinh(), casinhf(), casinhl()	Komplex arkusz koszinusz hiperbolikus.
catan(), catanf(), catanl()	Komplex arkusz tangens.
catanh(), catanhf(), catanhl()	Komplex arkusz tangens hiperbolikus.
ccos(), ccosf(), ccosl()	Komplex koszinusz.
ccosh(), ccoshf(), ccoshl()	Komplex koszinusz hiperbolikus.
cexp(), cexpf(), cexpl()	Komplex exponenciális függvény.
cimag(), cimagf(), cimagl()	Komplex szám képzetes része.
clog(), clogf(), clogl()	Komplex szám természetes alapú logaritmusa.
conj(), conjf(), conjl()	Komplex szám konjugáltja.
cpow(), cpowf(), cpowl()	Komplex szám hatványa.
cproj(), cprojf(), cprojl()	Vetítés a Riemann gömbre.
creal(), crealf(), creall()	Komplex szám valós része
csin(), csinf(), csinl()	Komplex szinusz.
csinh(), csinhf(), csinhl()	Komplex szinusz hiperbolikus.
csqrt(), csqrtf(), csqrtl()	Komplex négyzetgyök.
ctan(), ctanf(), ctanl()	Komplex tangens.
ctanh(), ctanhf(), ctanhl()	Komplex tangens hiperbolikus.

6.2.9 Általánosított típusú matematikai műveletek

Az <*tgmath.h*> deklarációs állomány beépíti a <*math.h*> és a <*complex.h*> include fájlokat, és típusfüggetlen nevű (*type-generic*) makrókat definiál a legtöbb matematikai függvényhez. A makró maga dönti el, hogy a hívási argumentum típusa alapján melyik C könyvtári függvényt kell aktiválnia.

Deklarációs állomány: *tgmath.h*

Az általánosított típusú makrók:

acos()	***acosh()***	***asin()***	***asinh()***	***atan()***
atan2()	***atanh()***	***cbrt()***	***ceil()***	***copysign()***
cos()	***cosh()***	***erf()***	***erfc()***	***exp()***
exp2()	***expm1()***	***fabs()***	***fdim()***	***floor()***
fma()	***fmax()***	***fmin()***	***fmod()***	***frexp()***
hypot()	***ilogb()***	***ldexp()***	***lgamma()***	***llrint()***
llround()	***log()***	***log10()***	***log1p()***	***log2()***
logb()	***lrint()***	***lround()***	***nearbyint()***	***nextafter()***
nexttoward()	***pow()***	***remainder()***	***remquo()***	***rint()***
round()	***scalbln()***	***scalbn()***	***sin()***	***sinh()***
sqrt()	***tan()***	***tanh()***	***tgamma()***	***trunc()***
carg()	***cimag()***	***conj()***	***cproj()***	***creal()***

A fenti makrók csak valós és komplex típusú argumentumokkal működnek helyesen, más típusok esetén az eredmény nem definiált.

6.2.10 Szabványos adatbevitel és -kivitel

Deklarációs állomány: *stdio.h*

Függvény	*Leírás*
vfscanf()	Formázott adatok olvasása adatfolyamból ***va_list*** típusú argumentum használatával.
vscanf()	Formázott adatok olvasása ***stdin***-ről ***va_list*** típusú argumentum használatával.
vsscanf()	Formázott adatok olvasása karaktersorozatból ***va_list*** típusú argumentum használatával.

A széles karakterekre épülő I/O függvényeket a megfelelő **char** típusú műveletekhez rendelve soroljuk fel. A kétféle adatfolyam között az ***fwide()*** függvény segítségével válthatunk.

Deklarációs állomány: *wchar.h*

Típusok és konstansok: ***wchar_t***, ***size_t***, ***wint_t***, ***NULL***, ***WCHAR_MAX***, ***WCHAR_MIN***, ***WEOF***

wchar_t *függvény*	***char*** *típusú megfelelő*
fgetwc()	***fgetc()***
fgetws()	***fgets()***
fputwc()	***fputc()***
fputws()	***fputs()***
fwide()	Az adatfolyam átalakítása ***wchar_t*** ↔ **char**
fwprintf()	***fprintf()***
fwscanf()	***fscanf()***
getwc()	***getc()***
getwchar()	***getchar()***
putwc()	***putc()***
putwchar()	***putchar()***
swprintf()	***sprintf()***
swscanf()	***sscanf()***
ungetwc()	***ungetc()***
vfwprintf()	***vfprintf()***
vfwscanf()	***vfscanf()***
vswprintf()	***vsprintf()***
vswscanf()	***vsscanf()***
vwprintf()	***vprintf()***
vwscanf()	***vscanf()***
wprintf()	***printf()***
wscanf()	***scanf()***

6.2.11 Pontos méretű egészek

A definiált típusok és makrók segítik a pontos (8, 16, 32, 64 bit) méretű egészek használatát.

Deklarációs állomány: *stdint.h*

Típusok	*Leírás*
int8_t, ***uint8_t***, ***int16_t***, ***uint16_t***, ***int32_t***, ***uint32_t***, ***int64_t***, ***uint64_t***	Pontos méretű egész típusok.
int_least8_t, ***uint_least8_t***, ***int_least16_t***, ***uint_least16_t***, ***int_least32_t***, ***uint_least32_t***, ***int_least64_t***, ***uint_least64_t***	Minimális méretű egész típusok.
int_fast8_t, ***uint_fast8_t***, ***int_fast16_t***, ***uint_fast16_t***, ***int_fast32_t***, ***uint_fast32_t***, ***int_fast64_t***, ***uint_fast64_t***	A leggyorsabb, minimális méretű egész típusok.
intmax_t, ***uintmax_t***	A legnagyobb méretű egész típusok.
intptr_t, ***uintptr_t***	Mutatók tárolására használható egész típusok.

A deklarációs fájl makrókat is definiál a fenti egész típusokhoz, melyek többsége az értékhatárokat tárolja. A felsorolásuk helyett a makrók nevében kis, dőlt *n* betűvel jelöljük a méretszám (8, 16, 32, 64) helyét. A *_C* végű makrók a megadott konstans értéket ellátják a típusjelző utótagokkal (*U*, *L*)

INTn_MIN, ***INTn_MAX***, ***UINTn_MAX***
INT_LEASTn_MIN, ***INT_LEASTn_MAX***, ***UINT_LEASTn_MAX***
INT_FASTn_MIN, ***INT_FASTn_MAX***, ***UINT_FASTn_MAX***
INTPTR_MIN, ***INTPTR_MAX***, ***UINTPTR_MAX***
INTMAX_MIN, ***INTMAX_MAX***, ***UINTMAX_MAX***
PTRDIFF_MIN, ***PTRDIFF_MAX***
SIG_ATOMIC_MIN, ***SIG_ATOMIC_MAX***
SIZE_MAX
WCHAR_MIN, ***WCHAR_MAX***
WINT_MIN, ***WINT_MAX***
INTn_C*(érték)*, ***UINTn_C****(érték)*
INTMAX_C*(érték)* , ***UINTMAX_C****(érték)*

6.2.12 Fomátum-konverziós előírások a pontos méretű egész típusokhoz

A definiált makrók segítik a pontos (*n*: 8, 16, 32, 64 bit) méretű egészek kiírását és beolvasását. A deklarációs állomány magába építi az <*stdint.h*> fájlt.

Deklarációs állomány: *inttypes.h*

Típus és makrók: ***imaxdiv_t***

PRIdn	***PRIdLEASTn***	***PRIdFASTn***	***PRIdMAX***	***PRIdPTR***
PRIin	***PRIiLEASTn***	***PRIiFASTn***	***PRIiMAX***	***PRIiPTR***
PRIon	***PRIoLEASTn***	***PRIoFASTn***	***PRIoMAX***	***PRIoPTR***
PRIun	***PRIuLEASTn***	***PRIuFASTn***	***PRIuMAX***	***PRIuPTR***
PRIxn	***PRIxLEASTn***	***PRIxFASTn***	***PRIxMAX***	***PRIxPTR***
PRIXn	***PRIXLEASTn***	***PRIXFASTn***	***PRIXMAX***	***PRIXPTR***
SCNdn	***SCNdLEASTn***	***SCNdFASTn***	***SCNdMAX***	***SCNdPTR***
SCNin	***SCNiLEASTn***	***SCNiFASTn***	***SCNiMAX***	***SCNiPTR***
SCNon	***SCNoLEASTn***	***SCNoFASTn***	***SCNoMAX***	***SCNoPTR***
SCNun	***SCNuLEASTn***	***SCNuFASTn***	***SCNuMAX***	***SCNuPTR***
SCNxn	***SCNxLEASTn***	***SCNxFASTn***	***SCNxMAX***	***SCNxPTR***

Függvény	*Leírás*
imaxabs()	Egész abszolút értéke.
imaxdiv()	Egész osztás, a hányados és a maradék az ***imaxdiv_t*** tpusú struktúrában adódik vissza.
strtoimax()	Sztring konvertálása maximális méretű előjeles egész számmá.
strtoumax()	Sztring átalakítása maximális méretű előjel nélküli egésszé.
wcstoimax()	Széles karakteres sztring konvertálása maximális méretű előjeles egész számmá.
wcstoumax()	Széles karakteres sztring átalakítása maximális méretű előjel nélküli egésszé.

6.2.13 A lebegőponos környezet kezelése

Deklarációs állomány: *fenv.h*

Típusok, kivétel- és kerekítési makrók valamint egy pragma: ***fenv_t***, ***fexcept_t***
FE_DIVBYZERO, ***FE_INEXACT***, ***FE_INVALID***, ***FE_OVERFLOW***, ***FE_UNDERFLOW***, ***FE_ALL_EXCEPT***
FE_DOWNWARD, ***FE_TONEAREST***, ***FE_TOWARDZERO***, ***FE_UPWARD***, ***FE_DFL_ENV***
#pragma STDC FENV_ACCESS *on/off*

Függvény	*Leírás*
feclearexcept()	Adott kivétel törlése.
fegetenv()	Az aktuális lebegőpontos környezet lekérdezése.
fegetexceptflag()	Az aktuális állapotjelzők lekérdezése.
fegetround()	Az aktuális kerekítési mód lekérdezése.
feholdexcept()	Az aktuális lebegőpontos környezet elmentése, és a kivételek törlése.
feraiseexcept()	Adott kivétel kiváltása.
fesetenv()	A lebegőpontos környezet beálltása.
fesetexceptflag()	Az állapotjelzők beállítása.
fesetround()	A kerekítési mód beállítása.
fetestexcept()	Adott kivétel bekövetkeztének ellenőrzése.
feupdateenv()	Az elmentett lebegőpontos környezet visszaállítása a kivételek megőrzése mellett.

6.2.14 Több-bájtos/széles karakterek közötti konverziók

Az ANSI C az ázsiai képírásjelek tárolására bevezette a több-bájtos karakterek fogalmát. A több-bájtos és a széles karakterek közötti átalakításra függvények találhatók a C könyvtárban.

Deklarációs állomány: *stdlib.h*

Típusok és makró: ***wchar_t***, ***size_t***, ***MB_CUR_MAX***

Függvény	*Leírás*
mblen()	A következő több-bájtos karakter hossza.
mbstowcs()	Több-bájtos sztring átalakítása széles karakteres sztringgé.
mbtowc()	Több-bájtos karakter átalakítása széles karakterré.
wctomb()	Széles karakter átalakítása több-bájtos karakterré.
wcstombs()	Széles karakteres sztring átalakítása több-bájtos sztringgé.

Deklarációs állomány: *wchar.h*

Típusok és makró: ***mbstate_t, wint_t, wchar_t, WEOF***

Az alábbi függvények többsége egy ***mbstate_t*** típusú állapotparamétertől függően a megfelelő fenti függvényeket hívják (*restartable*). Ha ez a paraméter nullaértékű, saját, belső ***mbstate_t*** objektumot használnak.

Függvény	*Leírás*
btowc()	Egybájtos karakter átalakítása a széles karakteres megfelelővé.
mbrlen()	Az ***mblen()*** függvényt bizonyos feltételek mellett hívó változat.
mbrtowc()	Az ***mbtowc()*** függvényt bizonyos feltételek mellett hívó változat.
mbsinit()	Igaz értékkel tér vissza, ha az argumentum egy kiindulási konverziós állapotra mutat.
mbsrtowcs()	Az ***mbstowcs()*** függvényt bizonyos feltételek mellett hívó változat.
wcrtomb()	A ***wctomb()*** függvényt bizonyos feltételek mellett hívó változat.
wctob()	Széles karakter átalakítása a megfelelő egybájtos karakterré.
wcsrtombs()	A ***wcstombs()*** függvényt bizonyos feltételek mellett hívó változat.

6.2.15 Kilépés a C99 programból

Deklarációs állomány: *stdlib.h*

Függvény	*Leírás*
_Exit()	Normál kilépés a programból a kilépési függvények meghívása és a szignálok aktiválása nélkül.

Deklarációs állomány: *assert.h*

Függvény	*Leírás*
assert()	C99-ben a hibaüzenet az ***assert()*** makró hívását tartalmazó függvény nevét is megjeleníti.

6.2.16 Az egész típusok méretadatai

Deklarációs állomány: *limits.h*

Makró	*Leírás*	*Határérték*
LLONG_MIN	**long long int** típusú objektum legkisebb értéke.	`-9223372036854775808LL`
LLONG_MAX	**long long int** típusú objektum legnagyobb értéke.	`+9223372036854775807LL`
ULLONG_MAX	**unsigned long long int** típusú objektum legnagyobb értéke.	`+18446744073709551615ULL`

6.2.17 A lebegőpontos típusok jellemző adatai

Deklarációs állomány: *float.h*

Makró	*Leírás*	*Érték (példa)*
DECIMAL_DIG	A decimális jegyek száma.	*21*
FLT_EVAL_METHOD	A lebegőpontos műveletek elvégzésének módja (*0: nincs kijelölt mód, 1: float→double, 2: float, double→long double, -1: definiálatlan*)	*0*

6.2.18 A logikai műveletek alternatív nevei

Deklarációs állomány: *iso646.h*

Makró	Jelentés
and	&&
and_eq	&=
bitand	&
bitor	|
compl	~
not	!
not_eq	!=
or	||
or_eq	|=
xor	^
xor_eq	^=

7. A szabványos C11 nyelv új lehetőségeinek áttekintése

A C11 nyelvet néhány új kulcsszó bevezetése, valamint a C könyvtár új típusokkal, felsorolásokkal, makrókkal és függvényekkel való bővítése jellemzi. A nyelv új kulcsszavai és fejállományai:

Új kulcsszavak	**Új include fájlok**
_Alignas	<*stdalign.h*>
_Alignof	<*stdatomic.h*>
_Atomic	<*stdnoreturn.h*>
_Generic	<*threads.h*>
_Noreturn	<*uchar.h*>
_Static_assert	
_Thread_local	

Az új szabvány biztonságos programozást segítő megoldásainak köszönhetően majdnem minden korábbi deklarációs állományban (<*stdlib.h*>, <*stdio.h*> <*string.h*> stb.) történt valamilyen változás.

7.1 A többszálú programok készítésének támogatása

Az új <*threads.h*> fejállomány a szálak, a mutexek és a feltételes változók létrehozását és vezérlését segítő függvényeket deklarál, valamint tartalmazza az **_Atomic** típusminősítőt.

Makrók: ***thread_local*, *ONCE_FLAG_INIT*, *TSS_DTOR_ITERATIONS*.**

Típusok: ***cnd_t*, *thrd_t*, *tss_t*, *mtx_t*, *tss_dtor_t*, *thrd_start_t*, *once_flag* .**

Felsorolás konstansok: ***mtx_plain*, *mtx_recursive*, *mtx_timed*, *thrd_timedout*, *thrd_success*, *thrd_busy*, *thrd_error*, *thrd_nomem*.**

Függvények a feltételes változók kezeléséhez:

```
void call_once(once_flag *flag, void (*func)(void));
int cnd_broadcast(cnd_t *cond);
void cnd_destroy(cnd_t *cond);
int cnd_init(cnd_t *cond);
int cnd_signal(cnd_t *cond);
int cnd_timedwait(cnd_t *restrict cond, mtx_t *restrict mtx, const struct timespec *restrict ts);
int cnd_wait(cnd_t *cond, mtx_t *mtx);
```

Mutex függvények:

```
void mtx_destroy(mtx_t *mtx);
int mtx_init(mtx_t *mtx, int type);
int mtx_lock(mtx_t *mtx);
int mtx_timedlock(mtx_t *restrict mtx, const struct timespec *restrict ts);
int mtx_trylock(mtx_t *mtx);
int mtx_unlock(mtx_t *mtx);
```

Szálkezelő függvények:

```
int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
thrd_t thrd_current(void);
int thrd_detach(thrd_t thr);
int thrd_equal(thrd_t thr0, thrd_t thr1);
_Noreturn  void thrd_exit(int res);
int thrd_join(thrd_t thr, int *res);
int thrd_sleep(const struct timespec *duration, struct timespec *remaining);
void thrd_yield(void);
```

Szálspecifikus tárolási függvények:

```
int tss_create(tss_t *key, tss_dtor_t dtor);
void tss_delete(tss_t key);
void *tss_get(tss_t key);
int tss_set(tss_t key, void *val);
```

Az <*stdatomic.h*> az objektumok megszakításmentes elérését biztosító eszközöket deklarál. Mindezeken kívül a C11 bevezeti a **_Thread_local** (***thread_local***) tárolási osztályt, amellyel megjelölt változókról minden szál saját másolattal rendelkezik.
Az alábbi példában elindítunk 7 szálat, majd a megállítás előtt mindegyik futását felfüggesztjük 3 másodpercig:

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

#define NSZALAK 7

static int szalSorszam[NSZALAK];

int SzalFuggveny(void * data) {
  struct timespec ido;
  printf("%d. szal inditasa...\n", *(int*)data);
  ido.tv_sec = 3;
  thrd_sleep(&ido, NULL);
  printf("%d. szal felebredt...\n", *(int*)data);
  return 0;
}

int main(void) {
  thrd_t szalID[NSZALAK];
  // Száladatok inicializálása
  for (int i=0; i < NSZALAK; ++i)
    szalSorszam[i] = i+1;
  // NSZALAK számú szál indítása
  for (int i=0; i < NSZALAK; ++i) {
    if (thrd_create(szalID+i, SzalFuggveny, &szalSorszam[i]) != thrd_success) {
      printf("%d. szal sikertelen inditasa.\n", i+1);
      return i+1;
    }
  }
  // Várakozás minden szál leállására
  for (int i=0; i < NSZALAK; ++i)
    thrd_join(szalID[i], NULL);
  return 0;
}
```

```
1. szal inditasa...
2. szal inditasa...
3. szal inditasa...
4. szal inditasa...
5. szal inditasa...
6. szal inditasa...
7. szal inditasa...
2. szal felebredt...
5. szal felebredt...
3. szal felebredt...
7. szal felebredt...
6. szal felebredt...
1. szal felebredt...
4. szal felebredt...
```

7.2 Atomi típusok és műveletek

Az új <*stdatomic.h*> *include* fájl több makrót, típust és függvényt deklarál, mely függvényekkel a szálak által közösen használt adatokon atomi műveleteket végezhetünk.

A fejállomány minden C11 egész típushoz az **_Atomic** típusminősítő segítségével atomi típusokat definiál, például:

```
typedef _Atomic(unsigned long long) atomic_ullong;
```

A ***memory_order*** felsorolás a szokásos (nem atomi) memória-szinkronizálási műveletek konstansait tartalmazza, míg az ***atomic_flag*** struktúra a zárolás-mentes, primitív atomi jelzők típusa.

Megtaláljuk itt az ***ATOMIC_BOOL_LOCK_FREE***, ***ATOMIC_CHAR_LOCK_FREE*** stb. atomi zárolás-mentes makrókat, az ***atomic_flag*** típusú objektumok inicializálására szolgáló ***ATOMIC_FLAG_INIT*** valamint az atomi objektumok inicializálását segítő ***ATOMIC_VAR_INIT*** makrót.

A programszálakban alkalmazható atomi függvények:

Szinkronizációs primitívek (fences) kezelése:
void atomic_thread_fence(memory_order order);
void atomic_signal_fence(memory_order order);

Az atomi típusok zárolás-mentes tulajdonságának lekérdezése:
_Bool ***atomic_is_lock_free***(***const volatile*** A *obj);

Az alábbi függvények mindegyike (az ***atomic_init*** kivételével) rendelkezik az alapfüggvény nevét ***_explicit*** taggal kiegészítő nevű függvénypárral, melyek egy vagy két ***memory_order*** típusú paraméterén keresztül a memória-szinkronizációról is rendelkezhetünk. (Az ***atomic_store()*** függvény esetén minkét formát megadjuk.)

Atomi típusokon végzett műveletek:

*void **atomic_init**(**volatile** A *object, C value);*
***void** **atomic_store**(**volatile** A *object, C desired);*
***void** **atomic_store_explicit**(**volatile** A *object, C desired, memory_order order);*
*C **atomic_load**(**volatile** A *object);*
*C **atomic_exchange**(**volatile** A *object, C desired);*
*_Bool **atomic_compare_exchange_strong**(**volatile** A *object, C *expected, C desired);*
*_Bool **atomic_compare_exchange_weak**(**volatile** A *object, C *expected, C desired);*
*C **atomic_fetch_key**(**volatile** A *object, M operand);*

Az atomi jelzőkre vonatkozó műveletek:

*_Bool **atomic_flag_test_and_set**(**volatile** atomic_flag *object);*
***void** **atomic_flag_clear**(**volatile** atomic_flag *object);*

Programrészlet az elmondottak illusztrálására:

```
atomic_int x = ATOMIC_VAR_INIT(12);
atomic_int y;
atomic_init(&y, 23);

// 1. programszál
int v1 = atomic_load_explicit(&x, memory_order_relaxed);
atomic_store_explicit(&y, v1, memory_order_relaxed);

// 2. programszál
int v2 = atomic_load_explicit(&y, memory_order_relaxed);
atomic_store_explicit(&x, v2, memory_order_relaxed);
```

7.3 Gyors, biztonságos kilépés a C programból

A C11 szálakkal is kapcsolatos sajátossága az ***stdlib.h***-ban deklarált

***_Noreturn void** **quick_exit**(**int** status);*

függvény, amely megkerüli a hagyományos ***exit()*** függvény hívásakor végrehajtódó kilépési folyamatot. A ***quick_exit()*** először meghívja az ugyancsak új

***int** **at_quick_exit** (**void** * func (**void**));*

hívásával regisztrált függvényeket, majd pedig aktivizálja az ***_Exit()*** függvényt, amely például nem üríti ki a fájlpuffereket, azonban ellátja a többszálúságból eredő kiléptetési feladatokat.

7.4 A _Noreturn függvényleíró

A **_Noreturn** kulcsszó hatására a fordító garantálja, hogy a megadott függvénynek nincs visszatérési értéke. Ezzel elkerülhetjük bizonyos fordítási üzenetek megjelenítését, valamint lehetővé tehetjük a fordító számára a visszatérési érték nélküli függvényekkel kapcsolatos optimalizálási lépések elvégzését.

***_Noreturn** **void** fugveny();*

Az <*stdnoreturn.h*> fejállomány a **noreturn** makrót definiálja a **_Noreturn** kulcsszóra:

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

// definiálatlan állapot, ha i <= 0, és kilép, ha i > 0
noreturn void Kilepes(int i) {
    if (i > 0) exit(i);
}
```

```
int main(void) {
  puts("Kilepes...");
  Kilepes(123);
  puts("Ez nem jelenik meg.");
}
```

A C11 szabványos könyvtár függvényei közül az alábbiak nem térnek vissza a hívás helyére:

***abort()*, *exit()*, *_Exit()*, *quick_exit()*, *thrd_exit()*, *longjmp()*.**

7.5 Változók és típusok memóriahatárra igazítása

A C11 nyelv a C++11 nyelvhez hasonló megoldásokat biztosít a változók memóriahatárra való igazítására. Az **_Alignas** és **_Alignof** kulcsszavak használatát itt is az <*stdalign.h*> állományban definiált ***alignas*** és ***alignof*** makrók segítik. További makrók (***__alignas_is_defined***, ***__alignof_is_defined***) 1 értékkel informálnak a fenti megoldások megvalósításáról.

Az ***alignas*** makróval kérhetjük a változók megadott határra való igazítástát:

```
alignas(double) int tomb[12];

alignas(32) double dadat;
```

A típusok esetén alkalmazott kiigazításról az ***alignof*** makró segítségével szerezhetünk információt:

```
printf("%llu\n", alignof(char));    // 1
printf("%llu\n", alignof(wchar_t)); // 2
printf("%llu\n", alignof(double));  // 8
```

Az <*stdlib.h*> fejállományban deklarált

***void** *aligned_alloc(size_t igazítás, size_t méret);*

függvény lefoglal adott méretű memóriaterületet, a megadott bájthatárra igazítva. (A méretet az igazítás egész számú többszöröseként kell megadnunk). A lefoglalt terület a ***free()*** vagy a ***realloc()*** hívással szabadítható fel.

Az <*stddef.h*> tartalmazza a ***max_align_t*** típus implementációfüggő definícióját. A típusból megtudhatjuk, hogy milyen határra igazítja az adott fordító a skaláris típusú változókat (ennél kisebb nem adható meg):

```
size_t a = alignof(max_align_t);
printf("A max_align_t igazítása %u (%#x)-os bajthatarra.\n", a, a); // 16
```

7.6 Névtelen struktúra és unió tagok

A C11 nyelv a C++ nyelvvel megegyező módon támogatja, hogy a **struct** és a **union** definíciókban névtelen struktúrákat és uniókat helyezzünk el. Ekkor ezek adattagjait közvetlenül, külön minősítés (.) nélkül érhetjük el.

```
struct TAdat {
  int m;
  union { // névtelen union
    char * kulcs;
    int index;
  };
};

struct {
  struct { // névtelen struct
    int azonosito;
    double meret;
  };
} stvar;
```

```
int main(void) {
  struct TAdat s1, s2;
  s1.kulcs = "Qwerty";
  s2.index = 129723;

  stvar.azonosito = 0x135791;
  stvar.meret = 45.67;
}
```

7.7 Az unicode kódolás bővített támogatása

A C11 nyelv az <*uchar.h*> fejállományban deklarált típusokkal (**char16_t**, **char32_t**) és függvényekkel segíti az unicode kódolású szövegek feldolgozását. A **char16_t** az UTF-16, míg a **char32_t** az UTF-32 kódoláshoz készült (az UTF-8 kódolás esetén továbbra is a **char** típust használjuk). Az új ***u*** és ***U*** előtagokkal unicode sztringkonstansokat, míg az ***u8*** előtaggal UTF-8 sztringliterált adhatunk meg.

```
#include <uchar.h>

// UTF-8
  char u8str[] = u8"π UTF-8 sztring violinkulcs: \U0001D11E, pi:\u03C0"

// UTF-16
#ifdef __STDC_UTF_16__
  char16_t u16str[] = u"€ UTF-16 sztring violinkulcs: \U0001D11E, pi:\u03C0";
  char16_t u16char  = u'€';
#endif

// UTF-32
#ifdef __STDC_UTF_32__
  char32_t u32str[] = U"♬ UTF-32 sztring violinkulcs: \U0001D11E, pi:\u03C0";
  char32_t u32char  = U'♬';
#endif
```

A fejállományban deklarált függvények a 16- és 32-bites széles karakterek (*c16*, *c32*) valamint a nulla-végű sztringben tárolt, többbájtos (*mb*) karakterek közötti konverziók elvégzésére szolgálnak:

*size_t **mbrtoc16**(char16_t * **restrict** pc16, **const char** * **restrict** s, size_t n, mbstate_t * **restrict** ps);*
*size_t **c16rtomb**(**char** * **restrict** s, char16_t c16, mbstate_t * **restrict** ps);*
*size_t **mbrtoc32**(char32_t * **restrict** pc32, **const char** * **restrict** s, size_t n, mbstate_t * **restrict** ps);*
*size_t **c32rtomb**(**char** * **restrict** s, char32_t c32, mbstate_t * **restrict** ps);*

7.8 Típusfüggő kifejezések (type-generic expressions)

A **_Generic** kulcsszó lehetővé teszi, hogy egy vezérlő kifejezés típusa alapján több kifejezés közül egyet kiválasszunk a fordítás folyamán.

_Generic *(vezérlő kifejezés, típus1 : kif1, … típus$_n$: kif$_n$,* **default***: kifdef)*

Amennyiben a *default* részt elhagyjuk, és a *vezérlő kifejezés* típusa egyik megadott *típussal* sem kompatibilis, fordítási hibát kapunk.

A megoldás emlékeztet a C++ nyelvben alkalmazott függvénynevek túlterhelésére, azonban itt nemcsak függvénynevek, hanem tetszőleges kifejezések is szerephetnek a kifejezések között. A megoldást gyakran makróba ágyazva használjuk.

```
#include <stdio.h>
#define TipusIndex(T) _Generic( (T), char: 1, int: 2, long: 3, default: 0)
int main() {
   printf("%d\n", _Generic( 'x', char: 1, int: 2, long: 3, default: 0)); // 2
   printf("%d\n", TipusIndex(7));   // 2
   printf("%d\n", TipusIndex("x")); // 0
}
```

A következő példában megadott *sinus* makró hívásakor az argumentum típusának fügvényében más és más könyvári függvény aktiválódik:

```
#define sinus(x) _Generic((x), long double: sinl, \
                               default: sin, \
                               float: sinf) (x)
```

7.9 A _Static_assert kulcsszó

A C11 fordító lehetővé teszi bizonyos ellenőrzések elhelyezését a fordító által feldolgozott kódban. A fordításidejű ellenőrzésekre használható a **_Static_assert** kulcsszó (szemben az ***assert*** makróval, amely futásidejű vizsgálatokra szolgál):

```
_Static_assert(konstans egész kifejezés, sztring);
```

A fordító kiértékeli a *konstans kifejezést*, és megjeleníti a *sztring* üzenetet, ha a kiértékelés eredménye 0.

Az alábbi példában azt ellenőrizzük, hogy a processzor típusa kisebb-e mint 6:

```
#define CPU 7
_Static_assert(CPU <= 5, "Nem tamogatott CPU");
```

A következő fordítási hibaüzenetet kapjuk: *„static assertion failed: "Nem tamogatott CPU"*. Az <*assert.h*> fejállomány a ***static_assert*** makróval fedi le a **_Static_assert** kulcsszót.

7.10 C11-specifikus előre definiált szimbolikus konstansok

A C11 szabvány az alábbi makrók definiálását javasolja a fordítóprogramoknak:

__STDC_VERSION__	Értéke *201112L* a C11 szabványú fordító esetén.
__STDC_UTF_16__	1, ha a **char16_t** típusú értékekek UTF-16 kódoslásúak.
__STDC_UTF_32__	1, ha a **char32_t** típusú értékekek UTF-32 kódoslásúak.
__STDC_ANALYZABLE__	1, ha a fordító teljesíti a C11 szabvány L (*Analyzability*) függelékében szereplő előírásokat.
__STDC_LIB_EXT1__	Értéke *201112L*, ha a fordító teljesíti a C11 szabvány K (*Bounds-checking interfaces*) függelékében szereplő előírásokat.
__STDC_NO_ATOMICS__	1, ha a fordító nem támogatja az atomi típusokat (beleértve az **_Atomic** típusmódosítót) és az <*stdatomic.h*> fejállományt.
__STDC_NO_COMPLEX__	1, ha a fordító nem biztosítja a komplex típusokat vagy a <*complex.h*> fejállományt.
__STDC_NO_THREADS__	1, ha a fordító nem tartalmazza a <*threads.h*> fejállományt.
__STDC_NO_VLA__	1, ha a fordító nem támogatja a változó méretű tömböket.

7.11 A complex típusú számok előállítása

A C99 kiegészítéseként bevezetett ***CMPLX*** makrókkal egyszerűen készíthetünk lebegőpontos complex típusú számokat. A makrók három változata a <*complex.h*> fejállományban:

double *complex* ***CMPLX****(****double*** *x, double y);*
float *complex* ***CMPLXF****(****float*** *x,* ***float*** *y);*
long *double complex* ***CMPLXL****(****long double*** *x,* ***long double*** *y);*

A makrók használatát bemutató példaprogram:

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

int main(void) {
    double complex a = CMPLX(12.0,-23.1);
    float complex  b = CMPLXF(7.29, 35);
    printf("a = %.2f%+.2fi\n", creal(a), cimag(a));
    printf("b = %.2f%+.2fi\n", creal(b), cimag(b));
}
```

```
a = 12.00-23.10i
b = 7.29+35.00i
```

7.12 Kizárólagos fájlelérés a létrehozás után – fopen()

A C11 szabványos könyvtárában találhat ***fopen()*** függvény (<*stdio.h*>) paraméterezésében a fájlelérési módok (*r*, *w* vagy *a*) kibővültek az *x* (kizárólagos létrehozás és megnyitás) móddal. Ezt az írásra való megnyitást előíró karaktersorozat végére kell illeszteni:

wx	szöveges állomány létrehozása írása kizárólagos (nem megosztott) eléréssel,
wbx	bináris fájl létrehozása írása kizárólagos eléréssel,
w+x	szövegfájl létrehozása írása/olvasásra kizárólagos eléréssel,
w+bx vagy *wb+x*	bináris állomány létrehozása írása/olvasásra kizárólagos eléréssel.

Amennyiben a létrehozni kívány állomány már létezik, vagy nem készíthető el, az *fopen()* hívás *NULL* mutatóval tér vissza.

7.13 A gets() függvény törlése a C könyvtárból

Már a C99 szabvány elavultnak nyilvánította a ***gets()*** – a szabványos inputról szöveget beolvasó – függvényt (<*stdio.h*>), azonban a C11 el is távolította azt a C szabványos könyvtárból. Ennek oka, hogy a ***gets()*** hívást tartalmazó program felhasználója az adatbevitel során felülírhatta a program adatterületének egy részét.

A C99 szabvány az ***fgets()*** függvény alkalmazását javasolta helyette, mely hívásakor korlátozhatjuk a beolvasandó karatersorozat hosszát:

***char** *fgets(**char** *__restrict__ str, **int** count, FILE *__restrict__ stream);*

A függvény legfeljebb *count–1* számú karatert olvas be, és a karatersorozatot nullás bájttal zárja. Sikeres olvasás esetén az *str* mutatóval, míg sikertelen esetben *NULL* értékkel tér vissza. Az eredmény sztring az <*Enter*> bilentyűnek megfelelő karaktert ('*n*') is tartalmazza, ha az belefér a megadott pufferbe.

```
#include <stdio.h>

int main(void) {
  char nev[12+1];
  fgets(nev, sizeof(nev), stdin);
  printf("\"%s\"\n", nev);
}
```

A C11 szabvány a ***gets_s()*** függvény bevezetésével egy még biztonságosabb megoldást javasol:

***char** *gets_s(**char** *str, rsize_t n);*

A ***gets_s()*** függvény az ***fgets()***-hez hasonlóan működik, azonban a beolvasott karakterek közé nem kerül be az <*Enter*>-nek megfelelő karakter. Ha a megadott szöveg túl hosszú, akkor üres sztring lesz a beolvasás eredménye:

```
#define __STDC_WANT_LIB_EXT1__ 1 // lásd a következő részt
#include <stdio.h>

int main(void) {
  char nev[12+1];
  gets_s(nev, sizeof(nev));
  printf("\"%s\"\n", nev);
}
```

7.14 A biztonságossá tett szabványos C könyvtár

A biztonsági rések keletkezésének a legfontosabb oka a C/C++ programokban a puffer túlcsordulása. Több fordítófejlesztő már évekkel korábban igyekezett biztoságos függvényekkel helyettesíteni a hagyományos könyvtári függvényeket. Példa erre a *Miscrosoft Visual Studio 2005*, ahol *_s* (*secure*: biztonságos) utótaggal látták el a nem biztonságosnak minősített könyvtári függggvények megbízhatóbb változatait. A C11 ehhez hasonló módon készítette el a C nyelv határellenőrző felületét (*bounds-checking interface*), amelyet a szabvány K függeléke taglal.

A hagyományos könyvtári függvények mellett csak akkor érhetők el a biztonságos függvények, ha a program elején, az *#include* sorok előtt 1 értékkel definiáljuk a *__STDC_WANT_LIB_EXT1__* szimbólumot:

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stddef.h>
#include <string.h>
#include <stdio.h>
```

7.14.1 Biztonságos bájtos puffer- és sztringkezelő függvények – <string.h>

A biztonságos függvények közül csupán néhányat mutatunk be részletesen, szemléltetve ezzel a C11 szabvány K függelékében leírtakat. Vegyük példaként a <*string.h*> fejállományban deklarált ***strcpy()*** függvényt, melynek prototípusa:

***char** *strcpy(**char** *restrict* dest, ***const char** *restrict* src);*

A függvény az *src* címen található sztringet a *dest* címen kezdődő területre másolja, majd pedig visszatér a *dest* mutatóval. A másolást az *src* sztring végének eléréséig folytatja, még akkor is, ha nincs elég hely a *dest* területen, vagy ha a *dest* mutató értéke ***NULL***.

Ezzel szemben a biztonságos ***strcpy_s()*** függvény teljesen biztonságosan használható, hisz mind a célterület méretét (*destsz*), mind pedig a célmutató nem nulla voltát is ellenőrzi. A függvény ***errno_t*** típusú visszatérési értéke pedig tudósít a művelet sikerességéről (*0*).

*errno_t **strcpy_s**(**char** *restrict* dest, rsize_t destsz, ***const char** *restrict* src);*

Amennyiben a célterület mérete nem teszi lehetővé a forrás karaktersorozat bemásolását (a 0-ás záró bájttal együtt) (*destsz<=strlen(src)*), üres sztring lesz a művelet eredménye, és az visszatérési érték jelzi ezt (*34*). A cél- és a forrásterület átfedettsége esetén szintén üres string lesz a másolás eredménye (*34*). Más hibakódot (*22*) kapunk, ha a *dest* ***NULL***, illetve ha a *destsz* 0 értékű vagy nagyobb, mint az *RSIZE_MAX* konstans értéke, és ekkor semmi sem másolódik a célterületre.

A felhasznált ***errno_t*** típust (**int**) az <*errno.h*>, az ***rsize_t*** típust (***size_t***) az <*stddef.h*>, míg az *RSIZE_MAX* szimbólumot az <*stdint.h*> állományok definiálják.

Az ***strcpy_s()*** függvényhez hasonlóan biztonságossá tett könyvtári függvényeket az alábbi formában használhatjuk:

```
char puffer[80+1];
errno_t err;
err = strcpy_s(puffer, sizeof(puffer), "forrasterulet");
if (err != 0)
{
    /* hibakezelés */
}
```

Példaként egy sztringbe szavakat másolunk az ***strcpy_s()*** és az ***strcat_s()*** függvényekkel. (Ez utóbbi működésének részletes leírása megtalálható a szabványban, illetve a referenciaként megadott weboldalon.) Amikor minden rendben van:

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
#include <stdio.h>

int main( void )
{
  char str[22+1];
  errno_t e1, e2, e3;
  #ifdef __STDC_LIB_EXT1__
    e1 = strcpy_s( str, sizeof(str), "Eleje " );
    printf( "szring:\"%s\"\n", str);
    e2 = strcat_s( str, sizeof(str), "kozepe " );
    printf( "szring:\"%s\"\n", str);
    e3 = strcat_s( str, sizeof(str), "vege." );
    printf( "szring:\"%s\"\n", str);
    printf("%d %d %d\n", e1, e2, e3);
  #endif
}
```

```
szring:"Eleje "
szring:"Eleje kozepe "
szring:"Eleje kozepe vege."
0 0 0
```

Amikor az utolsó lépésben túl hosszú szöveget szeretnénk hozzáfűzni:

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
#include <stdio.h>

int main( void )
{
  char str[22+1];
  errno_t e1, e2, e3;
  #ifdef __STDC_LIB_EXT1__
    e1 = strcpy_s( str, sizeof(str), "Eleje " );
    printf( "szring:\"%s\"\n", str);
    e2 = strcat_s( str, sizeof(str), "kozepe " );
    printf( "szring:\"%s\"\n", str);
    e3 = strcat_s( str, sizeof(str), " itt a vege fuss el vele." );
    printf( "szring:\"%s\"\n", str);
    printf("%d %d %d\n", e1, e2, e3);
  #endif
}
```

```
szring:"Eleje "
szring:"Eleje kozepe "
szring:""
0 0 34
```

További biztonságos puffer- és sztringkezelő függvények a <*string.h*>-ban:

errno_t ***memcpy_s****(**void** * **restrict** s1, rsize_t s1max, **const void** * **restrict** s2, rsize_t n);*
errno_t ***memmove_s****(**void** *s1, rsize_t s1max, **const void** *s2, rsize_t n);*
errno_t ***memset_s****(**void** *s, rsize_t smax, **int** c, rsize_t n);*

errno_t ***strcpy_s****(**char** * **restrict** s1, rsize_t s1max, **const char** * **restrict** s2);*
errno_t ***strncpy_s****(**char** * **restrict** s1, rsize_t s1max, **const char** * **restrict** s2, rsize_t n);*
errno_t ***strcat_s****(**char** * **restrict** s1, rsize_t s1max, **const char** * **restrict** s2);*
errno_t ***strncat_s****(**char** * **restrict** s1, rsize_t s1max, **const char** * **restrict** s2, rsize_t n);*
size_t ***strnlen_s****(**const char** *s, size_t maxsize);*
char** *****strtok_s*(**char** * **restrict** s1, rsize_t * **restrict** s1max, **const char** * **restrict** s2, **char** ** **restrict** ptr);*
errno_t ***strerror_s****(**char** *s, rsize_t maxsize, errno_t errnum);*
size_t ***strerrorlen_s****(errno_t errnum);*

Megjegyezük, hogy az ***strtok_s()*** függvény statikus puffer helyett, a *ptr* paraméterrel kijelölt memóriát használja munkaterületként.

7.14.2 A futásidejű előírások megsértésének központi kezelése – <stdlib.h>

A futtató rendszer számára beállítható, hogy mit tegyen, ha az *_s* utótagú függvények hívásakor sérülnek a biztonsági előírások. Az <*stdlib.h*> fejállományban deklarált ***set_constraint_handler_s()*** függvény segítségével beállíthatunk egy ***constraint_handler_t*** típusú kezelő függvényt. Választhatunk az előre definiált függvények közül, vagy saját függvényt is készíthetünk. A szabvány által biztosított

ignore_handler_s függvény semmit sem tesz, és általában ezt állítják be a különböző fordító implementációk alapértelmezettnek. Ezzel szemben az ***abort_handler_s*** kiír egy implementációfüggő hibaüzenetet, és hívja az ***abort()*** függvényt.

Saját kezelőfüggvény készítésekor paraméterben kapjuk meg a hibaüzenet mutatóját (*msg*), egy az implementáció által definiált objektum mutatóját vagy egy NULL mutatót (*ptr*), valamint a hiba kódját (*error*):

```
void SajatKezelo(const char *restrict msg, void *restrict ptr,  errno_t error)
{
  printf("Hibauzenet : %s\n", msg);
  printf("Mutato :     %#0x\n", ptr);
  printf("Hibakod :    %d\n", error);
}
```

A következő példában először nem teszünk semmit, majd pedig megszakítjuk a program futását, ha megsértjük a biztonsági előírásokat:

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main( void ) {
  char str[22+1];
  #ifdef __STDC_LIB_EXT1__
    set_constraint_handler_s(ignore_handler_s);
    strcpy_s( str, sizeof(str), "Eleje " );
    set_constraint_handler_s(abort_handler_s);
    strcat_s( str, sizeof(str), "kozepe " );
    strcat_s( str, sizeof(str), "vege." );
    printf( "szring:\"%s\"\n", str);
  #endif
}
```

További biztonságos megoldások az <*stdlib.h*>-ban:

constraint_handler_t ***set_constraint_handler_s****(constraint_handler_t handler);*
void abort_handler_s*(****const char*** *** restrict*** *msg,* ***void*** *** restrict*** *ptr, errno_t error);*
void ignore_handler_s*(****const char*** *** restrict*** *msg,* ***void*** *** restrict*** *ptr, errno_t error);*
errno_t ***getenv_s****(size_t * **restrict** len,* ***char*** *** restrict*** *value, rsize_t maxsize,* ***const char*** *** restrict*** *name);*
void *****bsearch_s****(****const void*** *key,* ***const void*** *base, rsize_t nmemb, rsize_t size,*
*****int*** *(*compar)(****const void*** *k,* ***const void*** *y,* ***void*** *context),* ***void*** *context);*
errno_t ***qsort_s****(****void*** *base, rsize_t nmemb, rsize_t size,*
*****int*** *(*compar)(****const void*** *x,* ***const void*** *y,* ***void*** *context),* ***void*** *context);*
errno_t ***wctomb_s****(****int*** *** restrict*** *status,* ***char*** *** restrict*** *s, rsize_t smax,* ***wchar_t*** *wc);*
errno_t ***mbstowcs_s****(size_t * **restrict** retval,* ***wchar_t*** *** restrict*** *dst, rsize_t dstmax,* ***const char*** *** restrict*** *src, rsize_t len);*
errno_t ***wcstombs_s****(size_t * **restrict** retval,* ***char*** *** restrict*** *dst, rsize_t dstmax,* ***const wchar_t*** *** restrict*** *src, rsize_t len);*

7.14.3 Biztonságos I/O kezelés – <stdio.h>

A biztonságos formázott adatkiviteli függvények esetén nem használható a ***%n*** formátum-előírás, mivel azzal a memóriába lehet írni, hiszen megkapjuk a formátumjelig kiírt karakterek számát.

errno_t ***tmpfile_s****(FILE * **restrict** * **restrict** streamptr);*
errno_t ***tmpnam_s****(****char*** *s, rsize_t maxsize);*
errno_t ***fopen_s****(FILE * **restrict** * **restrict** streamptr,* ***const char*** *** restrict*** *filename,* ***const char*** *** restrict*** *mode);*
errno_t ***freopen_s****(FILE * **restrict** * **restrict** newstreamptr,* ***const char*** *** restrict*** *filename,*
const char *** restrict*** *mode, FILE * **restrict** stream);*
int ***fprintf_s****(FILE * **restrict** stream,* ***const char*** *** restrict*** *format, ...);*
int ***fscanf_s****(FILE * **restrict** stream,* ***const char*** *** restrict*** *format, ...);*
int ***printf_s****(****const char*** *** restrict*** *format, ...);*
int ***scanf_s****(****const char*** *** restrict*** *format, ...);*

```
int snprintf_s(char * restrict s, rsize_t n, const char * restrict format, ...);
int sprintf_s(char * restrict s, rsize_t n, const char * restrict format, ...);
int sscanf_s(const char * restrict s, const char * restrict format, ...);
int vfprintf_s(FILE * restrict stream, const char * restrict format, va_list arg);
int vfscanf_s(FILE * restrict stream, const char * restrict format, va_list arg);
int vprintf_s(const char * restrict format, va_list arg);
int vscanf_s(const char * restrict format, va_list arg);
int vsnprintf_s(char * restrict s, rsize_t n, const char * restrict format, va_list arg);
int vsprintf_s(char * restrict s, rsize_t n, const char * restrict format, va_list arg);
int vsscanf_s(const char * restrict s, const char * restrict format, va_list arg);
char *gets_s(char *s, rsize_t n);
```

7.14.4 Biztonságos időkezelő függvények a <time.h>-ban

Az alábbi időkezelő függvények esetén biztonsági előírás a megfelelő pufferméret megadása, valamint az, hogy az évnek a [0, 9999] intervallumba kell esnie.

```
errno_t asctime_s(char *s, rsize_t maxsize, const struct tm *timeptr);
errno_t ctime_s(char *s, rsize_t maxsize, const time_t *timer);
struct tm *gmtime_s(const time_t * restrict timer, struct tm * restrict result);
struct tm *localtime_s(const time_t * restrict timer, struct tm * restrict result);
```

7.14.5 Biztonságos műveletek széles karakterekkel – <wchar.h>

A biztonságos formázott adatkiviteli függvények esetén itt sem használható a **%n** formátum-előírás, mivel azzal a memóriába lehet írni, hiszen megkapjuk a formátumjelig kiírt széles karakterek számát.

```
int fwprintf_s(FILE * restrict stream, const wchar_t * restrict format, ...);
int fwscanf_s(FILE * restrict stream, const wchar_t * restrict format, ...);
int snwprintf_s(wchar_t * restrict s, rsize_t n, const wchar_t * restrict format, ...);
int swprintf_s(wchar_t * restrict s, rsize_t n, const wchar_t * restrict format, ...);
int swscanf_s(const wchar_t * restrict s, const wchar_t * restrict format, ...);
int vfwprintf_s(FILE * restrict stream, const wchar_t * restrict format, va_list arg);
int vfwscanf_s(FILE * restrict stream, const wchar_t * restrict format, va_list arg);
int vsnwprintf_s(wchar_t * restrict s, rsize_t n, const wchar_t * restrict format, va_list arg);
int vswprintf_s(wchar_t * restrict s, rsize_t n, const wchar_t * restrict format, va_list arg);
int vswscanf_s(const wchar_t * restrict s, const wchar_t * restrict format, va_list arg);
int vwprintf_s(const wchar_t * restrict format, va_list arg);
int vwscanf_s(const wchar_t * restrict format, va_list arg);
int wprintf_s(const wchar_t * restrict format, ...);
int wscanf_s(const wchar_t * restrict format, ...);

errno_t wcscpy_s(wchar_t * restrict s1, rsize_t s1max, const wchar_t * restrict s2);
errno_t wcsncpy_s(wchar_t * restrict s1, rsize_t s1max, const wchar_t * restrict s2, rsize_t n);
errno_t wmemcpy_s(wchar_t * restrict s1, rsize_t s1max, const wchar_t * restrict s2, rsize_t n);
errno_t wmemmove_s(wchar_t *s1, rsize_t s1max, const wchar_t *s2, rsize_t n);
errno_t wcscat_s(wchar_t * restrict s1, rsize_t s1max, const wchar_t * restrict s2);
errno_t wcsncat_s(wchar_t * restrict s1, rsize_t s1max, const wchar_t * restrict s2, rsize_t n);
wchar_t *wcstok_s(wchar_t * restrict s1, rsize_t * restrict s1max, const wchar_t * restrict s2, wchar_t ** restrict ptr);
size_t wcsnlen_s(const wchar_t *s, size_t maxsize);
errno_t wcrtomb_s(size_t * restrict retval, char * restrict s, rsize_t smax, wchar_t wc, mbstate_t * restrict ps);
errno_t mbsrtowcs_s(size_t * restrict retval, wchar_t * restrict dst, rsize_t dstmax,
                    const char ** restrict src, rsize_t len, mbstate_t * restrict ps);
errno_t wcsrtombs_s(size_t * restrict retval, char * restrict dst, rsize_t dstmax,
                    const wchar_t ** restrict src, rsize_t len, mbstate_t * restrict ps);
```

Megjegyezük, hogy a ***wstrtok_s()*** függvény statikus puffer helyett, a *ptr* paraméterrel kijelölt memóriát használja munkaterületként.

7.15 Új előrások az implementációk számára a szabvány L függelékében

Az C11 szabvány L függeléke (*Analyzability*) a C program elemezhetőségének elősegítését célozza. Megnevezi egy kis részhalmazát a definiálatlan viselkedéseknek, és megad néhány implementációs előírást annak érdekében, hogy enyhítse a potenciális problémákat. Amennyiben a fordító implementációja megfelel ezeknek az előírásokat, 1 értékkel kell definiálnia az *__STDC_ANALYZABLE__* szimbólumot.