

A C nyelv alapvető adattípusai

Miért kell megadni az adat típusát?

Egy mérnöki program különböző adatokat kezelhet: például munkadarabok számát, hőmérsékletet, feszültséget vagy egy berendezés bekapcsolt állapotát. Ezekhez nem mindig ugyanaz az adattípus célszerű.

Az **adattípus** meghatározza, hogy milyen értékeket tárolhatunk egy változóban, mennyi memóriát foglal a változó, és hogyan vesz részt a műveletekben.

Tárolandó adat	Példa	Célszerű típus
Munkadarabok száma	120	int
Hőmérséklet °C-ban	-12.5	double
Feszültség V-ban	3.28	double
Egy egyszerű karakter	'A'	char
Bekapcsolt állapot	igaz vagy hamis	bool

A típus megválasztását a várható értéktartomány, a szükséges pontosság és a felhasználás együtt határozza meg.

Változó létrehozása és kezdőértéke

A **változó** egy névvel elérhető tárolóhely, amelynek értéke a program futása közben megváltozhat.

```
int darabszam = 120;  
double homerseklet = 23.5;  
char jeloles = 'A';
```

Az első sorban:

- az `int` a változó típusa;
- a `darabszam` a változó neve;
- a `120` a kezdőérték;
- a pontosvessző lezárja a deklarációt.

Később új értéket adhatunk a változónak:

```
darabszam = 125;
```

Az `=` jel itt **értékadást** jelent. A jobb oldali értéket a bal oldali változóba helyezzük. Az egyenlőség vizsgálatára a C nyelvben a `==` operátor szolgál.

A függvényen belül létrehozott egyszerű helyi változónak használat előtt adjunk értéket! A kezdőérték nélküli helyi `int` változóról nem feltételezhetjük, hogy nulla.

Memória, bit és bájt

A memória bitekből épül fel. Egy bit két értéket vehet fel: 0-t vagy 1-et. A szokásos mai számítógépeken **1 bájt = 8 bit**.

Nyolc bitnek $2^8 = 256$ különböző mintázata van. Ugyanezt a 256 mintázatot különböző módokon értelmezhetjük: például 0 és 255 közötti számként, vagy -128 és 127 közötti számként.

A több memória általában nagyobb értéktartományt vagy nagyobb pontosságot tesz lehetővé, de önmagában nem mondja meg, hogyan értelmezzük az adatot.

Fontos C-sajátosság: a `sizeof(char)` mindig 1, de a C szabvány nem minden gépen követel 8 bites bájtot. A bájtonkénti bitek számát a `<limits.h>` fejléc `CHAR_BIT` állandója adja meg. Ebben a jegyzetben a szokásos, 8 bites bájttal dolgozó rendszereket vesszük alapul.

Egész típusok

Az egész típusok törtrész nélküli számokat tárolnak. Például munkadarabok, mérési minták vagy cikluslépések számlálására használhatók.

Előjeles és előjel nélküli számok

Az **előjeles** típus negatív és nemnegatív értékeket is tárolhat. Az **előjel nélküli**, azaz **unsigned** típus csak nemnegatív értékeket tárolhat, de azonos bitszám esetén nagyobb a legnagyobb tárolható értéke.

Nyolc bit esetén, a szokásos kettes komplementes ábrázolással:

Típus	Legkisebb érték	Legnagyobb érték	Különböző értékek száma
Előjeles, 8 bites egész	-128	127	256
Előjel nélküli, 8 bites egész	0	255	256

Az **unsigned** nem biztosít több bitet: a rendelkezésre álló bitmintázatokat más értéktartományhoz rendeli.

Gyakori egész típusok

Az alábbi méretek és tartományok gyakori megvalósításokat mutatnak, nem minden C-rendszerre érvényes előírások. Kisebb mikrovezérlőkön például az `int` 16 bites is lehet. A táblázat előjeles tartományai kettes komplementes ábrázolásra vonatkoznak.

Típus	Gyakori méret bájtban	Bitszám	Legkisebb érték	Legnagyobb érték	printf() formátum
signed char	1	8	-128	127	%hhd

Típus	Gyakori méret bájtban	Bitszám	Legkisebb érték	Legnagyobb érték	printf() formátum
unsigned char	1	8	0	255	%hhu
short	2	16	-32768	32767	%hd
unsigned short	2	16	0	65535	%hu
int	4	32	-2147483648	2147483647	%d
unsigned int	4	32	0	4294967295	%u
long	4 vagy 8	32 vagy 64	a compilertől függ	a compilertől függ	%ld
unsigned long	4 vagy 8	32 vagy 64	0	a compilertől függ	%lu
long long	8	64	-9223372036854775808	9223372036854775807	%lld
unsigned long long	8	64	0	18446744073709551615	%llu

Az `int` és a `signed int` ugyanaz a típus. Hasonlóan a `short` és a `signed short`, illetve a `long` és a `signed long` is azonos típusjelölések. Emiatt nem szükséges külön sorban megtanulni őket.

A `long` nem jelent automatikusan 64 bites számot. Gyakori 64 bites Windows-környezetben 32 bites, míg gyakori 64 bites Linux-környezetben 64 bites.

Az egyszerű `char` külön figyelmet igényel: előjelessége a compilertől függ. Ha kifejezetten kis előjeles vagy előjel nélküli számot szeretnénk, írjunk `signed char` vagy `unsigned char` típust.

Példa: darabszám és készletváltozás

```
int darabszam = 120;
int keszletvaltozas = -8;
unsigned int meresek_szama = 500;
```

A készletváltozás negatív is lehet, ezért ehhez előjeles típus szükséges.

A pozitívnak szánt mennyiséghez nem kötelező unsigned típust választani. Kezdő programokban az `int` sok számlálási feladathoz megfelelő. Az `unsigned` típus nem ellenőrzi helyettünk, hogy egy kivonás eredménye negatív lenne-e; az előjeles és előjel nélküli típusok keverése pedig meglepő eredményt okozhat.

Mi történik, ha nem fér el az eredmény?

Egy típusnak véges az értéktartománya. Az előjeles egész művelet túlcsoordulásának nincs a C nyelv által meghatározott eredménye. **Nem támaszkodhatunk arra, hogy a legnagyobb pozitív szám után a legkisebb negatív következik.**

Az előjel nélküli számítás és az előjel nélküli típusba alakítás körbeforduló szabályt követ. Ha például az `unsigned char` 8 bites:

```
unsigned char szamlalo = 255;
szamlalo = szamlalo + 1;
```

Ekkor a változó új értéke 0. Az összeadás itt előbb int típusban adja a 256-ot, majd az unsigned char típusba történő visszaalakítás eredménye 0 lesz.

Mérnöki programban a körbefordulás gyakran hibás számlálást eredményez. A típust ezért a várható legnagyobb érték és a köztes számítások alapján válasszuk ki.

Lebegőpontos típusok

Hosszúság, idő, feszültség vagy hőmérséklet tárolásakor gyakran törtrészre is szükségünk van. Erre használhatók a **lebegőpontos típusok**.

```
double hossz = 1.25;
double ido = 0.08;
double homerseklet = -12.5;
```

A C forráskódjában a tizedesjel pont, nem vessző.

Értéktartomány és pontosság

A két fogalom különbözik:

- Az **értéktartomány** azt mutatja meg, milyen kicsi és milyen nagy nagyságrendű számokat tudunk tárolni.
- A **pontosság** azt mutatja meg, hány értékes számjegyet tudunk megőrizni.

Az értékes számjegyeket nem csak a tizedesjel után számoljuk. Például az 1234.56 és a 0.00123456 szám egyaránt hat értékes számjegyet tartalmaz.

A „6 számjegy pontosság” nem azt jelenti, hogy mindig hat tizedesjegyet tárolunk.

Nagyobb nagyságrendnél a szomszédos ábrázolható számok távolabb vannak egymástól.

Gyakori lebegőpontos típusok

A float és a double alábbi adatai a széles körben használt IEEE 754 binary32 és binary64 formátumokra vonatkoznak. A C nyelv más megvalósításokat is megenged.

Típus	Gyakori méret bájtnban	Legkisebb pozitív normál érték, közelítőleg	Legnagyobb véges pozitív érték, közelítőleg	Jellemző decimális pontosság	printf() formátum		
float	4	1.175×10	-38	3.403×10	38	kb. 6–7 értékes számjegy	%f, %e, %g

Típus	Gyakori méret bájtban	Legkisebb pozitív normál érték, közelítőleg	Legnagyobb véges pozitív érték, közelítőleg	Jellemző decimális pontosság	printf() formátum		
double	8	2.225×10	-308	1.798×10	308	kb. 15-16 értékes számjegy	%f, %e, %g
long double	megvalósításfüggő; gyakran 8, 12 vagy 16	megvalósításfüggő	megvalósításfüggő	legalább a double-é; gyakran kb. 18-19 vagy 33 számjegy	%Lf, %Le, %Lg		

Mindegyik típus tárolhat nullát és negatív értékeket is. Például a szokásos float legnegatívabb véges értéke közelítőleg -3.403×10^{38} .

A „legkisebb pozitív normál érték” **nem a legnegatívabb érték**, és nem is feltétlenül a legkisebb pozitív tárolható szám. Az IEEE 754 formátumok ennél kisebb pozitív, úgynevezett szubnormális számokat is tárolnak, csökkenő relatív pontossággal. A legkisebb pozitív binary32 érték körülbelül 1.401×10^{-45} , binary64 esetén körülbelül 4.941×10^{-324} .

A long double nem feltétlenül pontosabb a double típusnál. Egyes környezetekben azonos a két típus pontossága. A memóriában elfoglalt méretből önmagában nem állapítható meg a pontosság: a tárolás kitöltő bájtokat is tartalmazhat.

Melyiket használjuk mérnöki számításához?

Általános mérnöki számításoknál jó alapválasztás a double.

A float akkor lehet célszerű, ha sok mérési adatot tárolunk, fontos a memóriaigény, vagy a használt mikrovezérlő ezt támogatja hatékonyan. A long double speciális pontossági igény esetén lehet hasznos, a tényleges megvalósítás ellenőrzése után.

A számítás pontosságát nem csak a választott típus, hanem a mérési adatok minősége és az alkalmazott számítási módszer is befolyásolja.

Példa: mérési felbontás nagy értékénél

A szokásos 32 bites float az 100000000 környezetében már nem tud minden egész számot külön ábrázolni.

```
#include <stdio.h>

int main(void)
{
    float a = 100000000.0f;
    double b = 100000000.0;
```

```
a = a + 1.0f;  
b = b + 1.0;  
  
printf("float: %.0f\n", a);  
printf("double: %.0f\n", b);  
  
return 0;  
}
```

Szokásos IEEE 754 ábrázolással és a legközelebbi értékre kerekítéssel a kiírás:

```
float: 100000000  
double: 100000001
```

Az 1 nagyságú növekmény a float változóban elveszett. Mérnöki értelmezésben: ha nagy alapértéken kis változásokat szeretnénk mérni vagy összegezni, meg kell vizsgálnunk a számaábrázolás felbontását.

Az f utótag float típusú számállandót jelöl. Utótag nélkül az ilyen lebegőpontos számállandó double típusú. Például 1.5f típusa float, 1.5 típusa double.

Sok hétköznapi tizedes tört, például a 0.1, kettes számrendszerben nem írható le véges számú számjeggyel. Ezért lebegőpontos tároláskor kerekítésre lehet szükség. A double is véges pontosságú, és önmagában nem szünteti meg ezt a jelenséget.

A művelet típusa is számít

Egészosztás: egy gyakori hiba

Tegyük fel, hogy egy munkadarab 5 méter utat tesz meg 2 másodperc alatt.

```
int ut = 5;  
int ido = 2;  
double sebesseg = ut / ido;
```

A sebesseg értéke **2.0**, nem 2.5!

Ennek oka, hogy az osztás mindkét operandusa egész típusú, ezért **egészosztás** történik. A hányados törtrésze elmarad; pontosabban az eredmény nulla felé csonkolódik. A kapott 2 csak ezután alakul double típusú 2.0 értéké.

Az eredményt tároló változó típusa nem változtatja meg visszamenőleg a már elvégzett műveletet.

A helyes megoldás például:

```
double sebesseg = (double)ut / ido;
```

A (`double`) az ut értékét az osztás előtt `double` típusúvá alakítja. A művelet így lebegőpontos osztás lesz, az eredmény 2.5.

Ha a távolság és az idő mért mennyiség, célszerű már eleve lebegőpontos változókat használni:

```
double ut = 5.0;
double ido = 2.0;
double sebesseg = ut / ido;
```

Lebegőpontos érték átalakítása egészé

```
double homerseklet = 23.9;
int egesz_homerseklet = (int)homerseklet;
```

Az eredmény 23. Az átalakítás **nem a legközelebbi egészre kerekít**, hanem nulla felé csonkol: a -23.9 értékből -23 lesz. Ez akkor használható így, ha a csonkolt érték elfér a célként választott egész típusban.

Karakterek és logikai értékek

Karakter: `char`

A `char` egyetlen bájtnyi adatot tárol. Egyszerű karakterek kezelésére is használjuk:

```
char jeloles = 'A';
printf("Jeloles: %c\n", jeloles);
```

A karaktert **aposztrófok** közé írjuk. A `printf()` formátumszövegét és más szövegeket **idézőjelek** közé írjuk.

Egy `char` nem feltétlenül elegendő egy megjelenített karakter tárolására. UTF-8 kódolásban például sok ékezetes betű több bájtot foglal. A `char` ezért nem általános „egy Unicode-karakter” típus.

Logikai érték: `bool`

Igaz/hamis állapothoz C99-C17 környezetben a `<stdbool.h>` fejlécet használhatjuk:

```
#include <stdbool.h>

bool motor_bekapcsolva = true;
bool tulmelegedes = false;
```

A `true` igaz, a `false` hamis értéket jelent. A C feltételeiben a nulla hamisnak, a nem nulla érték igaznak számít. C23-tól a `bool`, `true` és `false` közvetlenül nyelvi kulcsszavak.

Értékek kiírása: printf()

A `printf()` használatához szükséges a `<stdio.h>` fejléc. A formátumjelzés megmondja, hogyan kell kiírni a hozzá tartozó argumentumot.

```
#include <stdio.h>

int main(void)
{
    int darabszam = 120;
    double feszultseg = 3.276;

    printf("Darabszam: %d\n", darabszam);
    printf("Feszultseg: %.2f V\n", feszultseg);

    return 0;
}
```

A kiírás:

```
Darabszam: 120
Feszultseg: 3.28 V
```

A `\n` új sort kezd. A `%.2f` két tizedesjegyet ír ki, a megjelenítéshez kerekítve az értéket. **Ez nem változtatja meg a változóban tárolt értéket.**

Formátum	Felhasználás
%d	int kiírása
%u	unsigned int kiírása
%ld	long kiírása
%lld	long long kiírása
%c	egy karakter kiírása
%f	float vagy double kiírása szokásos tizedes alakban
%.3f	float vagy double kiírása három tizedesjeggyel
%e	float vagy double kiírása tudományos jelöléssel
%g	a nagyságrendhez igazodó alak; alapértelmezésben hat értékes számjegy
%Lf	long double kiírása
%zu	sizeof eredményének kiírása

Például a `2.500000e-03` jelentése 2.5×10^{-3} , vagyis 0.0025.

A `printf()` hívásakor a float argumentum automatikusan double típusúvá alakul. Emiatt mindkettőhöz `%f` használható. A `printf()` esetén a `%lf` is megengedett, de ugyanazt jelenti, mint a `%f`; kezdőként elegendő a `%f` alakot használni.

A scanf() szabálya eltér: float változó beolvasásához `%f`, double változóhoz `%lf` szükséges. A kiírás és a beolvasás formátumjelzéseit ne keverjük össze!

Hogyan ellenőrizzük a tényleges méreteket?

A sizeof operátor megadja egy típus vagy változó méretét bájtban.

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

int main(void)
{
    printf("Bit/bajt: %d\n", CHAR_BIT);
    printf("int:          %zu bajt\n", sizeof(int));
    printf("long:         %zu bajt\n", sizeof(long));
    printf("float:         %zu bajt\n", sizeof(float));
    printf("double:        %zu bajt\n", sizeof(double));
    printf("long double: %zu bajt\n", sizeof(long double));

    printf("int minimum: %d\n", INT_MIN);
    printf("int maximum: %d\n", INT_MAX);
    printf("double maximum: %e\n", DBL_MAX);

    printf("float decimalis pontossaga: %d\n", FLT_DIG);
    printf("double decimalis pontossaga: %d\n", DBL_DIG);
    printf("long double decimalis pontossaga: %d\n", LDBL_DIG);

    return 0;
}
```

A <limits.h> az egész típusok határértékeiről, a <float.h> a lebegőpontos típusok tulajdonságairól ad információt. A FLT_DIG, DBL_DIG és LDBL_DIG a veszteségmentes decimális-bináris-decimális oda-vissza alakításhoz garantált értékes decimális számjegyek számát jellemzi, a megfelelő tartományban. A szokásos float esetén FLT_DIG = 6, a szokásos double esetén DBL_DIG = 15.

Kiegészítés: amikor pontos bitszám szükséges

Adatátvitelnél, fájlformátumoknál vagy mikrovezérlős feladatoknál előírás lehet a pontos bitszám. A <stdint.h> fejlécben ilyen típusnevek is elérhetők, ha a rendszer támogatja az adott pontos méretet:

```
#include <stdint.h>

uint8_t meresi_bajt = 200;
int16_t nyers_meres = -1200;
uint32_t impulzusok = 100000;
```

A uint8_t 8 bites előjel nélküli, az int16_t 16 bites előjeles, a uint32_t 32 bites előjel nélküli egész típust jelöl. Ezek kiírásához hordozható megoldást a <inttypes.h> formátummakrói adnak; a %d nem használható automatikusan minden ilyen típushoz.

Típusválasztás mérnöki feladatokban

Feladat	Jó kiinduló választás	Mire figyeljünk?
Darabszám, cikluslépések száma	int	Férjen el a legnagyobb várható érték.
Előjeles változás, eltérés	int vagy double	Szükséges-e törtrész?
Hossz, idő, erő, feszültség	double	Mértékegység, szükséges pontosság, kerekítési hiba.
Sok, korlátozott pontosságú mérési adat	float	Valóban elegendő-e a felbontása?
Nagy egész számláló	long long	A köztes műveletek is megfelelő típusban történjenek.
Egy adatbájt	unsigned char	A bájt bitszáma; adatátvitelnél a pontos formátum.
Egy egyszerű karakter	char	Többbájtos kódolás esetén nem elég egyetlen char.
Igen/nem állapot	bool	Egyértelmű változónév.

A változó típusa nem tárol mértékegységet. Ha `double hossz = 12.0;`, a program nem tudja, hogy ez méter vagy milliméter. A mértékegységet változónévvel és következetes használattal tegyük egyértelművé, például `hossz_m` vagy `hossz_mm`.

Önellenőrző kérdések

1. Miért célszerű a munkadarabok számát egész, a hőmérsékletet pedig lebegőpontos típusban tárolni?
2. Több különböző értéket tárol-e egy 8 bites előjel nélküli típus, mint egy 8 bites előjeles típus?
3. Mennyi lesz a `double x = 7 / 2;` utasításban az `x` értéke? Miért?
4. Miben különbözik a `7 / 2` és a `7.0 / 2` eredménye?
5. Mennyi lesz az `int x = (int)8.9;` utasításban az `x` értéke?
6. Mit jelent a `printf("%.2f", x);` formátumában a `.2`? Megváltozik-e az `x` értéke?
7. Biztosan 8 bájtos-e a `long` egy 64 bites számítógépen?
8. Miért veszhet el egy kis növekmény, ha nagyon nagy `float` értékhez adjuk?

Rövid válaszok

1. A darabszám törtrész nélküli mennyiség; a hőmérsékletnél gyakran törtrészt is meg kell őrizni.
2. Nem. Mindkettőnek 256 bitmintázata van; az értéktartományuk különbözik.
3. 3.0, mert előbb egészosztás történik.
4. Az első eredménye egész 3, a másodiké lebegőpontos 3.5.
5. 8, mert az átalakítás nulla felé csonkol.
6. Két tizedesjegyet ír ki; a tárolt érték nem változik meg.
7. Nem. A `long` mérete a compilertől függ.
8. A nagy érték környezetében az ábrázolható szomszédos számok közötti távolság meghaladhatja a növekményt.

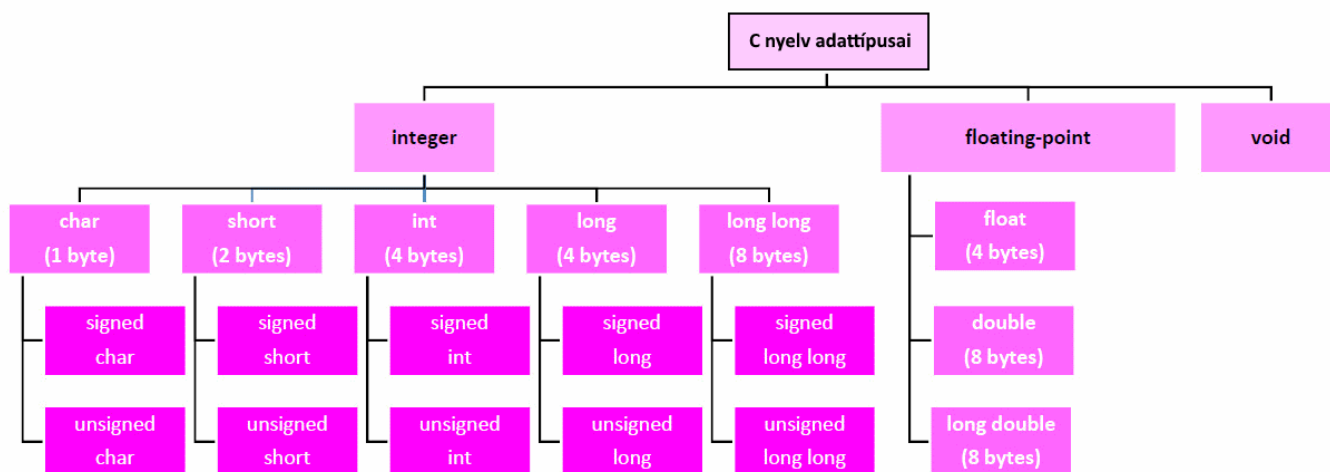
Oktatói megjegyzés és szakmai háttér

Ez a jegyzet az alapvető számszerű típusokra, valamint a karakter- és logikai értékekre összpontosít. A tömbök, szövegek, mutatók, struktúrák, felsorolások és a void külön tananyagrészen tárgyalhatók.

A példák C11/C17 környezetben használhatók. A méret- és pontosságadatok megvalósításfüggőségét a táblázatok jelzik. Az első órán a típusválasztásra, az egészosztásra és a double használatára érdemes összpontosítani; a szubnormális számok és a pontos bitszámú típusok kiegészítő ismeretként kezelhetők.

Szakmai háttér: [A C11 szabvány nyilvános bizottsági tervezete \(N1570\)](#), különösen az 5.2.4.2 (határértékek), 6.2.5 (típusok), 6.3 (típuskonverziók), 6.5.5 (osztás), 6.5.3.4 (sizeof), 7.18 (logikai értékek), 7.20 (egész típusok) és 7.21.6.1–2 (formázott kiírás és beolvasás) szakaszok.

Összefoglalás



From:

<https://edu.iit.uni-miskolc.hu/> - Institute of Information Science - University of Miskolc

Permanent link:

https://edu.iit.uni-miskolc.hu/tanszek:oktatas:szamitastechnika:adat_tipusok?rev=1792604198

Last update: 2026/10/21 17:36

