C NYELV - NAGY CHEAT SHEET / GYORSKALAUZ

Pythonról érkezőknek - az eredeti cheat sheet teljes tartalmával, nagy részletességgel kibővítve

Cél: legyen egyszerre gyors referencia és magyarázó jegyzet. Az eredeti 4 oldalas változat minden lényegi témája megmarad: C dialektusok, integer típusok, uint16_t, signed/unsigned, floatok, 32/64 bit, const/static, C string és NUL, char* vs char[], =/==/strcmp, strcpy/strncpy, konverziók, pointerek, int**, malloc/sizeof, UTF-8, Rust/Python összevetés. Ezeket nem lecseréljük, hanem kibontjuk.

Python -> C mentális váltás: Pythonban a runtime sok memóriarészletet elrejt. C-ben neked kell tudnod, milyen típusú objektum van egy címen, mekkora, meddig él, módosítható-e, ki birtokolja, és hány byte/elem érhető el.

Tartalom: 1) szabvány és build; 2) integer/floating típusok; 3) konverziók; 4) const/static; 5) tömbök; 6) C stringek; 7) másolás/összehasonlítás; 8) pointerek; 9) memória; 10) struct/enum/bool; 11) byte-szintű műveletek; 12) UTF-8; 13) hibák/UB; 14) API-tervezés; 15) Rust/Python kitekintés; 16) receptek és ellenőrzőlisták.

1. C dialektusok, szabványok és fordító

C89/C90: klasszikus régi alap. C99: jelentős modernizálás, többek között <stdint.h>, rugalmasabb deklarációk, // komment. C11: további nyelvi/könyvtári elemek. C17: főleg korrekciós szabvány. C23: a jelenlegi új generációs szabvány.

```
cc -std=c17 -Wall -Wextra -Wpedantic -Wconversion main.c -o main

# fejlesztéskor sok toolchainen:
cc -std=c17 -g -O1 -Wall -Wextra    -fsanitize=address,undefined main.c -o main
```

A -std=... választja a nyelvi szabályrendszert. A warningok nem ugyanazok, mint a fordítási hibák: a fordító gyakran enged olyan kódot is, ami helyes ugyan, de gyanús.

Szabvány vs platform: a C sok helyen csak minimumokat és szabályokat garantál. Például int nem kötelezően 32 bit. Az adott gépen a sizeof, <limits.h> és <stdint.h> mutatja a konkrétumokat.

Defined, implementation-defined, unspecified, undefined

Kategória	Mit jelent?
defined	A szabvány pontos viselkedést ír elő.
implementation-defined	A fordító/platform választ egy lehetőséget és dokumentálja.
unspecified	Több szabványos lehetőség közül választhat, de nem feltétlen dokumentálja.
undefined behavior (UB)	A szabvány semmilyen viselkedést nem ír elő; nem lehet rá támaszkodni.

UB nem „random eredmény”. A fordító optimalizálhat abból a feltételezésből, hogy UB nem történik. Ezért egy UB-s program akár debug buildben „működhet”, release-ben pedig teljesen másképp viselkedhet.

2. Mi az az int? Hányféle integer típus van?

int egy előjeles egész típus. A C integer típusrendszere nem egyetlen int-ből áll, hanem több szélességi családból és signed/unsigned változatból.

C típus	Minimum	Mai tipikus méret	Megjegyzés
signed char	legalább 8 bit	8	külön integer típus a sima char mellett
unsigned char	legalább 8 bit	8	nyers byte-okhoz különösen hasznos
short / short int	legalább 16 bit	16	signed alapértelmezéssel
unsigned short	legalább 16 bit	16	nemnegatív
int / signed int	legalább 16 bit	32	általános egész
unsigned int	ugyanazon rang	32	nemnegatív modulo aritmetika
long / long int	legalább 32 bit	32 vagy 64	platformfüggő
unsigned long	legalább 32 bit	32 vagy 64	platformfüggő
long long	legalább 64 bit	64	széles egész
unsigned long long	legalább 64 bit	64	széles unsigned

signed int ugyanaz, mint int. A long int röviden long; a long long int röviden long long.

signed és unsigned

signed: negatív értéket is reprezentál, unsigned: 0 és pozitív értékek; N bites unsigned tipikus tartománya 0..2^N-1.

Fontos: unsigned túlcsordulás definiált modulo aritmetika. Signed integer túlcsordulás UB. Emiatt ne használd az unsigned típust csak azért, mert „nagyobb pozitív szám kell”, ha közben signed/unsigned keverés lesz.

```
int a = -1;
unsigned int b = 1;

/* az összehasonlítás előtt konverziók történhetnek */
if (a < b) {
    /* nem biztos, hogy azt kapod, amit intuitívan vársz */
}
```

3. Fix szélességű integerek: int8_t, uint16_t, int32_t

A <stdint.h> szabványos header fix vagy minimum szélességű integer típusneveket ad. A helyes név uint16_t, nem uint_16t.

Típus	Jelentés
int8_t / uint8_t	pontosan 8 bites signed/unsigned, ha ilyen típus létezik
int16_t / uint16_t	pontosan 16 bit
int32_t / uint32_t	pontosan 32 bit
int64_t / uint64_t	pontosan 64 bit
int_least16_t	legalább 16 bites, legkisebb megfelelő típus
int_fast16_t	legalább 16 bites, implementáció szerint gyors típus
intptr_t / uintptr_t	képes pointerértéket integerként reprezentálni, ha rendelkezésre áll

```
#include <stdint.h>

uint8_t  flags = 0xA5u;
uint16_t port  = 443u;
int32_t  temp  = -1200;
uint64_t big   = UINT64_C(10000000000);
```

A uint8_t sok implementáción valójában unsigned char. Ez azért érdekes, mert stream/kiírás vagy integer promotion esetén char-jellegű részletekbe futhatsz.

size_t és ptrdiff_t

size_t a méretek természetes unsigned típusa; a sizeof ezt adja. Pointerkülönbséghez a szabványos signed típus ptrdiff_t.

```
size_t n = sizeof(int);
printf("%zu\n", n);
```

4. Integer literálok és jelölések

Forma	Példa	Jelentés
decimális	42	tízes számrendszer
oktális	052	nyolcas; a vezető 0 fontos
hex	0x2A	tizenhatos
bináris (C23)	0b101010	kettes
unsigned suffix	42U	unsigned
long suffix	42L	long
long long suffix	42LL	long long
kombinált	42ULL	unsigned long long

Oktális csapda: a 010 nem decimális tíz, hanem oktális 10 = decimális 8.

printf formátumok

Típus	printf
int	%d vagy %i
unsigned int	%u
long	%ld
unsigned long	%lu
long long	%lld
unsigned long long	%llu
size_t	%zu
pointer	%p, argumentum tipikusan (void*)p

Fix szélességű típusokra a <inttypes.h> PRI... makrói hordozhatóbbak.

```
#include <inttypes.h>
uint32_t id = 123;
printf("id=%" PRIu32 "\n", id);
```

5. 32 bit vs 64 bit - ILP32, LP64, LLP64

A „64 bites rendszer” nem azt jelenti, hogy minden C típus 64 bites. Az adatmodell mondja meg, hogy az alap integer és pointer típusok mekkorák.

Adatmodell	int	long	long long	pointer	Tipikus példa
ILP32	32	32	64	32	32 bites rendszerek
LP64	32	64	64	64	Linux/macOS 64 bit
LLP64	32	32	64	64	Windows 64 bit

Következmény: ha protokollban/fájlban pontosan 64 bites mező kell, ne azt írd, hogy „long, mert 64 bites gépen fut”. Használj pl. uint64_t-t.

```
printf("char=%zu short=%zu int=%zu long=%zu long long=%zu ptr=%zu\n",
       sizeof(char), sizeof(short), sizeof(int),
       sizeof(long), sizeof(long long), sizeof(void*));
```

6. float, double, long double

Típus	Gyakori méret	Tipikus pontosság	Literál
float	32 bit	kb. 6-9 decimális számjegy	1.5f
double	64 bit	kb. 15-17 számjegy	1.5
long double	64/80/96/128 bit	implementációfüggő	1.5L

A bináris lebegőpontos ábrázolás sok decimális törtet nem tud pontosan tárolni. Ezért 0.1 + 0.2 == 0.3 nem általánosan jó ellenőrzés.

```
#include <math.h>

double a = 0.1 + 0.2;
double b = 0.3;

if (fabs(a - b) < 1e-12) {
    /* közel azonos */
}
```

Nem minden == rossz floatnál. Ha például pontosan egy korábban beállított sentinel értéket vizsgálsz, lehet helye. Számítási eredményeknél viszont gyakran abszolút + relatív tolerancia kell.

NaN és infinity

Lebegőpontos környezetben lehet NaN és végtelen érték. NaN különleges: még saját magával sem egyenlő. Teszteléshez isnan, isfinite stb. használható.

7. Automatikus konverziók, integer promotion, cast

C-ben a kisebb integer típusok művelet közben gyakran int-té vagy unsigned int-té promotálódnak. Ezért a kifejezés típusa nem feltétlen ugyanaz, mint az operandusok deklarált típusa.

```
uint8_t a = 250;
uint8_t b = 10;
int sum = a + b;  // tipikusan int művelet
```

Numeric cast

```
double d = 3.9;
int i = (int)d;      // tört rész eldobása, ha eredmény reprezentálható

int x = 300;
uint8_t y = (uint8_t)x;  // 300 nem fér 8 bitbe
```

Unsigned cél esetén a konverzió modulo jellegű. Signed célba nem reprezentálható érték konverziója finomabb szabályú/implementációfüggő lehet; portábilis kódban előbb tartományt ellenőrizz.

A cast nem validáció. A (uint8_t)x nem ellenőrzi, hogy x 0..255 között van.

Pointer cast nem numeric cast

A pointer cast nem alakítja át a mögöttes objektumot. Csak azt mondja: ugyanazt a címet most más típusú pointerként kezelem. Dereferálásnál alignment, effective type/aliasing és méret miatt UB lehet.

8. const - mit jelent pontosan?

Deklaráció	Jelentés
const int x	x ezen a néven át nem módosítható
const char *p	p más címre állítható, de *p p-n keresztül nem módosítható
char * const p	p maga nem állítható más címre, de *p módosítható
const char * const p	sem p, sem a mutatott adat nem módosítható ezen az útvonalon

```
char buf[] = "abc";

const char *p = buf;
p++;          // OK
// *p = 'X';  // nem OK ezen az útvonalon

char * const q = buf;
*q = 'X';     // OK
// q++;       // nem OK
```

const nem ownership. Attól, hogy egy pointer const char*, még nem tudod, meddig él a memória, ki birtokolja vagy mekkora.

String literál és const

A C string-literál típusa történelmileg nem const char[], de a módosítása UB. Ezért modern API-használatban literálra tipikusan const char*-t használj.

9. static - három fő szerep

Lokális static

```
void f(void) {
    static int calls = 0;
    calls++;
}
```

A calls neve csak a függvényen belül látszik, de az objektum a program teljes futása alatt él és megőrzi az értékét.

Fájlszintű static változó

```
static int internal_state = 0;
```

Internal linkage: más fordítási egységből nem ugyanazon a néven hivatkozható.

static függvény

```
static int helper(int x) {
    return x * 2;
}
```

A helper név csak ebben a fordítási egységben látszik.

Static storage inicializálás: ha nincs explicit inicializáló, null-inicializálást kap. Ez eltér az automatikus lokális változótól.

10. Tömbök: char[10], int[10], valódi tömb vs pointer

```
int a[4] = {10, 20, 30, 40};

printf("%zu\n", sizeof a);       // teljes tömb byte-mérete
printf("%zu\n", sizeof a[0]);    // egy elem
printf("%zu\n", sizeof a / sizeof a[0]); // elemszám itt 4
```

A legtöbb kifejezésben a tömb neve az első elemre mutató pointerré alakul. Ez az úgynevezett array-to-pointer conversion.

```
int *p = a;
printf("%d\n", p[2]);      // 30
printf("%d\n", *(p + 2));  // 30
```

p[i] lényegében *(p+i). A pointeraritmetika a pointee típus méretével skálázódik.

Függvényparaméterben a [] félrevezető lehet

```
void f(int a[10]) {
    /* itt a paraméter lényegében int *a */
}
```

A függvény nem kap automatikusan 10-es kapacitásinformációt. Biztonságosabb API: void f(int *a, size_t count).

11. Mi az a C string? NUL, NULL, newline

A klasszikus C string egy NUL-lal lezárt char byte-sorozat. A NUL a numerikus 0 értékű byte, C literálban '\0'.

```
char s[] = "cat";
// memória: 'c' 'a' 't' '\0'
strlen(s) == 3
sizeof(s) == 4
```

Jelölés	Jelentés
'\0'	NUL byte, C string lezáró
NULL	null pointer konstans/makró; nem karakter
'\n'	newline / line feed
'\r'	carriage return
"\r\n"	CRLF, klasszikus DOS/Windows szöveges sorvég

A NUL nem a newline. Egy string tartalmazhat \n-t középen, és utána folytatódhat. A string végét a \0 jelöli.

Ha nincs NUL a hozzáférhető bufferben: strlen, strcmp, printf("%s") túlolvashat a buffer határán -> UB.

12. char* vs char[] vs char[10]

Forma	Mi ez?	Mit tudsz?
char *p	pointer char-ra	csak címet; kapacitást önmagában nem
char a[] = "cat"	4 elemű char tömb	módosítható saját tároló, sizeof(a)==4
char a[10] = "cat"	10 elemű char tömb	kapacitás 10 byte, aktuális stringhossz 3
const char *p = "cat"	pointer string-literálra	olvasásra kezeld

```
char a[] = "hello";        // 6 byte
char b[10] = "hello";      // 10 byte tárhely
const char *c = "hello";   // pointer literálra

a[0] = 'H';                // OK
b[5] = '!';                // a régi NUL felülírva
b[6] = '\0';               // új lezárás
```

8 karakteres string mekkora?

8 ASCII karakter = 8 byte adat + 1 byte NUL, tehát legalább 9 byte. UTF-8 esetén 8 felhasználói karakter több mint 8 byte is lehet.

Kapacitás és hossz két külön fogalom. char b[10]="hi": kapacitás 10, stringhossz 2, felhasznált byte a NUL-lal együtt 3.

13. String memóriaképe: literál, tömb, pointer

```
char a[] = "abc";
const char *p = "abc";

sizeof(a)   // 4
sizeof(p)   // pointer mérete, pl. 8
strlen(a)   // 3
strlen(p)   // 3
```

char a[]="abc" létrehoz egy saját tömböt. const char *p="abc" csak egy pointerváltozót hoz létre, amely egy literál tárolójára mutat.

sizeof vs strlen: sizeof a tárhely/típus méretét adja byte-ban. strlen futásidőben NUL-ig számol. Pointeren sizeof(p) nem a string hossza.

Embedded NUL

```
char weird[] = {'a', 'b', '\0', 'c', 'd', '\0'};

strlen(weird) == 2; // C string szempontból "ab"
```

A mögöttes tömb 6 byte, de a klasszikus string API az első NUL-nál megáll. Bináris adathoz ezért pointer + explicit hossz kell.

14. =, ==, strcmp - mit hasonlítunk?

= értékadás. == skalár értékegyenlőség. Integernél közvetlenül a számértékeket hasonlíthatod. Stringnél a változók gyakran pointerként jelennek meg, ezért == címeket hasonlít.

```
int a = 3;
int b = 3;

if (a == b) { /* integer érték azonos */ }

char s1[] = "abc";
char s2[] = "abc";

if (s1 == s2) { /* címek: nem stringtartalom */ }

if (strcmp(s1, s2) == 0) {
    /* stringtartalom azonos */
}
```

strcmp(x,y) 0, ha azonos; negatív, ha x lexikografikusan kisebb; pozitív, ha nagyobb. Ne feltételezd, hogy pontosan -1 vagy +1.

Gyakori kezdőhiba: if (x = 5) értékadás. Warningokkal a fordító sokszor segít felismerni.

15. strncmp, memcmp - hasonlítanak, de nem ugyanaz

strncmp(a,b,n) legfeljebb n karakterig/byte-ig hasonlít C string szemantikával. Ha hamarabb NUL jön, megáll. Prefixvizsgálathoz használható.

memcmp(a,b,n) pontosan n byte-ot hasonlít nyers memóriaként. Nem áll meg NUL-nál.

```
char a[] = "ab\0x";
char b[] = "ab\0y";

strcmp(a, b) == 0;       // mindkettő C stringként "ab"
memcmp(a, b, 4) != 0;    // a 4 byte eltér
```

memcmp nem struct-egyenlőség univerzálisan. Structban padding byte-ok lehetnek; azok tartalma miatt két mezőnként azonos struct byteképe nem feltétlen biztonságosan hasonlítható memcmp-pal.

16. strcpy miért veszélyes? strncpy miért nem egyszerű megoldás?

strcpy(dst,src) addig másol, amíg NUL-t nem talál. A célbuffer kapacitását nem kapja meg, így nem tudja ellenőrizni, hogy belefér-e.

```
char dst[5];
strcpy(dst, "elephant");  // overflow -> UB
```

strncpy(dst,src,n) más szabályú: ha a forrás legalább n hosszú, nem garantál lezáró NUL-t; ha rövidebb, nullákkal feltölti n byte-ig.

```
char dst[5];
strncpy(dst, "elephant", sizeof dst);
/* dst: e l e p h -- NUL nélkül */
```

strncpy != safe strcpy. Előbb döntsd el a kívánt szemantikát: hibázzon, ha nem fér bele? Csonkoljon? Fix szélességű mezőt töltsön?

Kapacitástudatos másolás

```
bool copy_string(char *dst, size_t cap, const char *src) {
    size_t len = strlen(src);
    if (len >= cap)
        return false;          // len + 1 NUL kell
    memcpy(dst, src, len + 1);
    return true;
}
```

17. memcpy, memmove, memset

Függvény	Feladat
memcpy(dst,src,n)	pontosan n byte másolása; átfedés esetén UB
memmove(dst,src,n)	pontosan n byte; átfedés esetén is definiált
memset(dst,value,n)	n byte feltöltése ugyanazzal a byte-értékkel

```
char s[] = "abcdef";
memmove(s + 1, s, 5); // átfedő tartomány; memmove kell
```

memset(p,0,n) byte-szinten nulláz. Integer nullázásra hétköznapi modern platformokon működik, de a fogalom byte-reprezentáció; ne kezeld úgy, mintha minden típus minden lehetséges platformon matematikai „0” objektummá válna ugyanettől.

memset gyakori hiba

```
int a[100];
memset(a, 1, sizeof a);
/* nem 100 darab integer 1 lesz!
   minden byte 0x01 lesz */
```

18. snprintf és biztonságosabb formázás

```
char buf[32];
int n = snprintf(buf, sizeof buf, "id=%d", id);

if (n < 0) {
    /* formázási hiba */
} else if ((size_t)n >= sizeof buf) {
    /* csonkolás történt */
}
```

A snprintf kapacitást kap. A visszatérési érték általában azt a karakterszámot jelzi, amely a teljes eredményhez kellett volna NUL nélkül.

Mekkora buffer kell egy formázott stringhez?

```
int needed = snprintf(NULL, 0, "%d:%s", id, name);
if (needed >= 0) {
    char *s = malloc((size_t)needed + 1);
    if (s)
        snprintf(s, (size_t)needed + 1, "%d:%s", id, name);
}
```

19. fgets és sorbeolvasás

Szöveges sor beolvasására fix bufferbe gyakori a fgets. A kapacitást megkapja, de a newline-ot megtarthatja.

```
char line[128];

if (fgets(line, sizeof line, stdin)) {
    size_t len = strlen(line);
    if (len > 0 && line[len - 1] == '\n') {
        line[len - 1] = '\0';
    }
}
```

Ha a sor hosszabb a buffernél: a teljes sor nem fér be egyetlen hívással. Ha ez számít, kezelni kell a maradékot vagy dinamikus stratégiát használni.

scanf("%s")

Szélességkorlát nélkül a scanf("%s",buf) túlírhatja a buffert. Ha használod, adj mezőszélességet, de összetett inputhoz gyakran egyszerűbb soronként olvasni és utána parse-olni.

20. Szöveg -> szám: strtol, strtoul, strtod

atoi egyszerű, de hibakezelése gyenge. Robusztusabb a strtol család.

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

errno = 0;
char *end = NULL;
long v = strtol(text, &end, 10);

if (end == text) {
    /* nem volt feldolgozható szám */
} else if (*end != '\0') {
    /* maradt extra karakter */
} else if (errno == ERANGE) {
    /* tartományhiba */
} else {
    /* v használható */
}
```

Python párhuzam: Pythonban az int(text) hibánál exceptiont dob. C-ben tipikusan visszatérési érték + end pointer + errno együtt adja a teljes állapotot.

21. Pointer alapok: cím (&) és dereferálás (*)

```
int x = 42;
int *p = &x;

printf("%d\n", x);   // 42
printf("%d\n", *p);  // 42

*p = 99;              // x most 99

printf("%p\n", (void*)p);   // x címe
printf("%p\n", (void*)&p);  // a pointerváltozó saját címe
```

Deklarációban int *p: p pointer int-re. Kifejezésben *p: dereferáld p-t, vagyis érd el a pointee objektumot.

A pointer nem az objektum. p egy külön változó, amely egy címszerű értéket tárol; ezért &p és p más címet jelent.

22. Pointerre mutató pointer: int **

```
int x = 10;
int *p = &x;
int **pp = &p;

**pp = 77;  // x = 77
*p = 88;    // x = 88
```

Ha p típusa int*, akkor &p típusa int**. A **pp kétszer dereferál.

Miért hasznos?

Ha egy függvénynek a hívó pointerváltozóját kell módosítania, pointer-to-pointer kellhet.

```
bool make_int(int **out) {
    *out = malloc(sizeof **out);
    if (*out == NULL)
        return false;

    **out = 123;
    return true;
}

int *p = NULL;
if (make_int(&p)) {
    printf("%d\n", *p);
    free(p);
}
```

23. Pointer aritmetika

```
int a[4] = {10, 20, 30, 40};
int *p = a;

p++;           // most a[1]-re mutat
printf("%d", *p); // 20
```

p+1 nem egy byte-tal lép, hanem egy int elemmel. A fordító a pointee típus méretével skáláz.

One-past-the-end

```
int *begin = a;
int *end = a + 4;

for (int *it = begin; it != end; ++it)
    printf("%d\n", *it);
```

Az end képezhető és összevethető, de nem dereferálható.

Pointeraritmetika határa: ugyanazon tömb/objektum tartományán belül értelmes. Tetszőleges, független objektumok címeivel végzett aritmetika nem általános numerikus címkezelés.

24. NULL, dangling pointer, use-after-free

```
int *p = malloc(sizeof *p);
if (p == NULL)
    return;

*p = 10;
free(p);

/* *p = 20; */   // use-after-free: UB
p = NULL;
```

NULL egy explicit „nem mutat objektumra” állapot. A dangling pointer viszont nem feltétlen NULL: lehet benne régi cím, amelynek az objektuma már megszűnt.

free(NULL) megengedett. Ugyanazt az allokációt kétszer felszabadítani viszont UB.

Alias pointer

```
int *p = malloc(sizeof *p);
int *q = p;
free(p);
p = NULL;
/* q továbbra is dangling */
```

25. 8 bites integer beírása int pointeren keresztül

```
uint8_t small = 200;

int x;
int *p = &x;

*p = small;    // értékkonverzió: 200 -> int, rendben
```

Itt az értéket olvassuk ki small-ból, majd intté konvertáljuk és valódi int objektumba írjuk.

A veszélves félreértés

```
uint8_t small = 200;
uint8_t *q = &small;

int *bad = (int*)q;
/* int x = *bad; */  // alignment, méret, aliasing probléma -> UB lehet
```

Pointer cast != adatkonverzió. Egy 1 byte-os objektumból nem lesz int objektum csak azért, mert a címét int*-re castolod.

26. Endianness: byte-okból integer

Többbyte-os integer memóriabeli byte-sorrendje platformfüggő lehet. Protokoll/fájlformátum esetén explicit kódold/dekódold a sorrendet.

```
uint8_t b[2] = {0x12, 0x34};

uint16_t v = ((uint16_t)b[0] << 8) |
             ((uint16_t)b[1]);

/* v = 0x1234, host endiannessétől független */
```

Ne használd vakon: *(uint16_t*)b. Endianness mellett alignment és aliasing problémát is okozhat.

27. malloc, sizeof és dinamikus allokáció

```
size_t n = 10;

int *a = malloc(n * sizeof *a);
if (a == NULL) {
    /* allokáció sikertelen */
}

/* a[0] ... a[n-1] */

free(a);
a = NULL;
```

malloc byte-ok számát kapja. Az eredmény nyers, inicializálatlan dinamikus tárhely.

Jó idiom: malloc(n * sizeof *a), nem malloc(n * sizeof(int)). Így ha a pointer típusa változik, a méretszámítás is együtt változik.

Szorzás túlcsordulása

```
if (n > SIZE_MAX / sizeof *a) {
    /* túl nagy kérés */
}
a = malloc(n * sizeof *a);
```

28. calloc, realloc, free

Függvény	Lényeg
malloc(n)	n byte inicializálatlan tárhely
calloc(count,size)	count*size byte, byte-szinten nullázva
realloc(p,new_size)	méretváltoztatás; az allokáció elmozdulhat
free(p)	dinamikus allokáció felszabadítása

Biztonságos realloc minta

```
int *tmp = realloc(a, new_count * sizeof *a);

if (tmp == NULL) {
    /* a még mindig a régi blokkra mutat */
} else {
    a = tmp;
}
```

Miért tmp? Ha közvetlenül a = realloc(a,...)-t írsz, sikertelenségnél elveszítheted a régi pointert.

29. Stack / automatikus tárhely, static storage, heap

Tárhely/élettartam	Példa	Meddig él?
automatikus lokális	int x; függvényben	blokk végéig
static storage	static int x; vagy globális	program végéig
heap/dinamikus	malloc	free-ig

```
int *bad(void) {
    int x = 5;
    return &x;  // x élettartama véget ér returnkor
}

int *ok(void) {
    int *p = malloc(sizeof *p);
    if (p) *p = 5;
    return p;   // caller free-zze
}
```

Dangling pointer a stackből is lehet. Nem csak free után lesz érvénytelen pointer; lokális objektum címének visszaadása is klasszikus hiba.

30. Mekkora hely kell egy stringnek? Dinamikus stringmásolat

```
const char *src = "12345678";

size_t len = strlen(src);     // 8
char *s = malloc(len + 1);    // +1 NUL

if (s != NULL) {
    memcpy(s, src, len + 1);
    /* s most saját módosítható másolat */
    free(s);
}
```

A strlen NUL előtti bytehosszt ad. A tároláshoz +1 byte kell a NUL-nak.

Ownership

A fenti s owned heap string: aki megkapja és megtartja, annak tudnia kell, hogy mikor kell felszabadítani.

C típusrendszer nem jelöli az ownershipet. Ugyanaz a char* lehet owned heap memória, borrowed buffer, static tároló vagy egy tömb belsejére mutató pointer. Dokumentáció/API-konvenció szükséges.

31. sizeof mélyebben

sizeof(T) a típus byte-méretét adja; sizeof expr az expression típusának méretét. Eredménytípusa size_t.

```
int a[10];

sizeof(int)
sizeof a
sizeof a[0]
sizeof a / sizeof a[0]
```

Normál esetben a sizeof(expr) nem értékeli ki az expressiont.

```
int i = 0;
size_t n = sizeof(i++);
/* i tipikusan változatlan marad */
```

Tömb vs pointer: sizeof a valódi tömbnél teljes tömbméret. Ha a tömb pointerré alakult, már csak pointerméretet kapsz.

32. bool C-ben

```
#include <stdbool.h>

bool ready = true;

if (!ready) {
    /* ... */
}
```

C-ben a 0 hamis, minden nemzero igaz feltételben. A bool kényelmes név a nyelvi logikai típus köré.

strcmp és bool csapda

```
if (strcmp(a, b)) {
    /* ez azt jelenti: NEM azonosak */
}

if (strcmp(a, b) == 0) {
    /* azonosak */
}
```

Miért? A strcmp 0-val jelzi az egyenlőséget, a C feltételben pedig 0 = false.

33. enum

```
enum State {
    STATE_IDLE,
    STATE_RUNNING,
    STATE_ERROR
};

enum State s = STATE_IDLE;
```

Az enum olvasható neveket ad egész konstansoknak. Állapotgépekhez, opciókhoz és kategóriákhoz jobb, mint „magic numberöket” szétszórni.

Explicit értékek

```
enum ErrorCode {
    ERR_OK = 0,
    ERR_IO = 10,
    ERR_PARSE = 20
};
```

Külső protokoll/bináris ABI esetén ne támaszkodj vakon az enum memóriaméretére; rögzített mezőhöz használj megfelelő integer típust és konverziót.

34. struct

```
struct User {
    int id;
    char name[32];
};

struct User u = {
    .id = 1,
    .name = "Ada"
};

struct User *p = &u;

printf("%d\n", u.id);
printf("%d\n", p->id);   // ugyanaz, mint (*p).id
```

Padding és alignment

A struct mérete lehet nagyobb a mezők byte-méretének egyszerű összegénél, mert a fordító padding byte-okat illeszthet be.

Bináris fájlra structot közvetlenül kiírni nem hordozható általános megoldás. Padding, endianness, integer méretek és ABI eltérhetnek.

35. union - röviden

```
union Number {
    int i;
    float f;
};
```

A union tagjai ugyanazt a tárhelyet osztják meg. Egyszerre a reprezentáció ugyanazon byte-jait használják. Alacsony szintű kódban hasznos lehet, de az aktív tag és reprezentáció kérdéseit érteni kell.

Type punning: unionon vagy pointer caston keresztüli reprezentációértelmezés szabályai finomak. Hordozható byte-szintű másoláshoz gyakran a memcpy a legkevésbé meglepő út.

36. Bitműveletek és flag-ek

```
enum {
    FLAG_READ  = 1u << 0,
    FLAG_WRITE = 1u << 1,
    FLAG_EXEC  = 1u << 2
};

unsigned flags = FLAG_READ | FLAG_WRITE;

if (flags & FLAG_WRITE) { /* beállítva */ }

flags |= FLAG_EXEC;      // beállítás
flags &= ~FLAG_WRITE;    // törlés
flags ^= FLAG_READ;      // kapcsolás
```

Operátor	Jelentés
&	bitwise AND
|	bitwise OR
^	XOR
~	bitwise NOT
<< / >>	shift
&& / || / !	logikai operátorok - nem ugyanazok

37. Byte buffer + explicit hossz

Nem minden adat string. Bináris protokoll, tömörített adat, kép vagy olyan szöveg, amely embedded NUL-t tartalmazhat, jobban modellezhető pointer + hossz párossal.

```
struct ByteSlice {
    const uint8_t *data;
    size_t len;
};

void dump(const uint8_t *data, size_t len) {
    for (size_t i = 0; i < len; ++i)
        printf("%02X ", data[i]);
}
```

Ez közelebb áll a Rust slice vagy Python bytes szemléletéhez. A vég nem sentinelből következik, hanem külön hosszadatból.

38. UTF-8: byte, kódpont, grapheme

UTF-8-ban három fogalmat különíts el: byte, Unicode kódpont, grapheme cluster (amit a felhasználó egy karakternek érzékelhet).

```
const char *s = "árvíz";
printf("%zu\n", strlen(s));
```

A strlen byte-okat számol. Magyar ékezetes betűk UTF-8-ban tipikusan többbyte-osak, ezért a bytehossz nagyobb lehet a látható betűk számánál.

s[1] a második byte. Nem garantáltan a második Unicode kódpont vagy második látható karakter.

Miért működik mégis a NUL lezárás UTF-8-nál?

Az UTF-8 nem használ 0 byte-ot többbyte-os kódpont részeként; a 0 byte az U+0000-hoz tartozik. Ezért a klasszikus NUL-lezárás együtt tud élni UTF-8 szöveggel.

39. UTF-8 összehasonlítás

Ha két string byte-szinten ugyanazt a normalizált UTF-8 szekvenciát tartalmazza, strcmp használható egyenlőségre. De emberi szöveghez további szemantikák kellenek.

- Normalization: vizuálisan azonos szöveg eltérő kódpont-sorozatban létezhet.
- Case folding: kis-/nagybetűs összevetés nem egyszerű byte-lowercase.
- Collation: nyelvi rendezési sorrend locale- és szabályfüggő.
- Grapheme: felhasználói karakter lehet több kódpontból álló klaszter.

Példa: az „é” lehet egyetlen U+00E9, vagy „e” + combining acute accent. Vizuálisan hasonló, byte-szinten eltérő.

Komoly Unicode összehasonlításhoz/normalizációhoz használj Unicode könyvtárat, például ICU-t vagy a platform megfelelő Unicode API-ját.

40. strcmp vs strcoll vs Unicode

strcmp byte-alapú lexikografikus sorrend. strcoll a C locale mechanizmusához kapcsolódó összehasonlítást adhat, de modern Unicode alkalmazásoknál a teljes nyelvi collation továbbra is külön könyvtári kérdés.

Ne keverd: „byte-szinten ugyanaz”, „Unicode szövegként kanonikusan ugyanaz” és „emberi nyelvi rendezésben ugyanott van” három külön kérdés.

41. Függvényparaméterek: C érték szerint ad át

```
void set_bad(int x) {
    x = 10;
}

void set_ok(int *x) {
    *x = 10;
}

int n = 1;
set_bad(n);   // n marad 1
set_ok(&n);   // n lesz 10
```

Pointer átadásakor is a pointerérték másolata kerül a függvénybe. A pointee azért módosulhat, mert a másolt pointer ugyanarra az objektumra mutat.

Out-paraméter

```
bool parse_number(const char *text, long *out) {
    char *end = NULL;
    errno = 0;
    long v = strtol(text, &end, 10);

    if (errno || end == text || *end != '\0')
        return false;

    *out = v;
    return true;
}
```

42. Function pointer és callback

```
int add(int a, int b) {
    return a + b;
}

int (*op)(int, int) = add;
int result = op(2, 3);
```

op pointer olyan függvényre, amely két intet kap és intet ad vissza. Callback API-kban gyakori.

Callback koncepció

```
void for_each(int *a, size_t n, void (*fn)(int)) {
    for (size_t i = 0; i < n; ++i)
        fn(a[i]);
}
```

43. Undefined behavior katalógus

Hiba	Példa
signed overflow	INT_MAX + 1
out-of-bounds	a[10] egy 10 elemű tömbnél
use-after-free	free(p); *p
double free	free(p); free(p)
NULL dereference	*p amikor p==NULL
string literal írás	const nélkül literálra írni
uninitialized read	int x; printf("%d",x)
hibás pointer cast + dereference	1 byte-os tárhely int*-ként dereferálva
túl nagy/hibás shift	shift count a típus szélességén kívül

ASan + UBSan: fejlesztéskor nagyon sok ilyen hibát futásidőben megfognak. Nem bizonyítják a helyességet, de erős hibakereső eszközök.

44. Inicializálás és indeterminált értékek

```
int x = 0;
int a[10] = {0};

struct Point {
    int x, y;
};

struct Point p = {0};
```

Automatikus lokális változó explicit inicializálás nélkül indeterminált értékkel indulhat. Sok esetben olvasása UB.

```
int x;
/* printf("%d", x);  // hiba */
```

Static storage objektum implicit null-inicializálást kap.

45. Strict aliasing / effective type - gyakorlati szinten

A fordító optimalizációja feltételezhet bizonyos típusú aliasing szabályokat. Ezért tetszőleges pointer casttal ugyanazokat a byte-okat más típusú objektumként dereferálni veszélyes.

```
float f = 1.0f;
/* uint32_t bits = *(uint32_t*)&f;  // problémás lehet */

uint32_t bits;
memcpy(&bits, &f, sizeof bits);     // hordozhatóbb byte-másolási minta
```

unsigned char / char byte-hozzáférés különleges. Objektum reprezentációjának byte-jai char-jellegű pointeren keresztül vizsgálhatók.

46. Alignment

Bizonyos típusú objektumokat a memóriában megfelelő címhatárra kell igazítani. Egy tetszőleges uint8_t* cím nem biztos, hogy megfelelően igazított int*-hez.

Ez még egy ok, amiért veszélyes: int *p=(int*)byte_pointer; *p. Lehet, hogy a cím nem megfelelően igazított.

A malloc által visszaadott tárhely megfelelően igazított a szokásos objektumtípusokhoz; egy buffer közepére mutató tetszőleges offset viszont már nem feltétlen.

47. C string API gyors referencia

Függvény	Mit csinál?	Fő kockázat/megjegyzés
strlen	NUL előtti bytehossz	érvényes lezárt string kell
strcmp	teljes string összevetés	byte-alapú
strncmp	max. n byte/karakter összevetés	prefixnél hasznos
strchr	karakter keresése	NUL-lezárt input
strstr	substring keresése	NUL-lezárt input
strcpy	másolás NUL-lal	célkapacitást nem tud
strncpy	max n másolás/padding	NUL nem garantált
strcat	hozzáfűzés	célkapacitást nem tud
snprintf	formázott írás kapacitással	return értéket ellenőrizni

48. mem* API gyors referencia

Függvény	Használat
memcpy	nyers byte-másolás, nem átfedő tartomány
memmove	nyers byte-másolás, átfedés is megengedett
memcmp	pontosan n byte összehasonlítása
memset	n byte feltöltése ugyanazzal a byte-értékkel

Ezek nem stringfüggvények: explicit bytehosszt kapnak és nem NUL-ig dolgoznak.

49. Makrók, const, inline

```
#define BUFFER_SIZE 128
static const size_t buffer_size = 128;

#define SQUARE(x) ((x) * (x))

static inline int square_int(int x) {
    return x * x;
}
```

A preprocessor makró szöveges helyettesítés. A függvény típusos. Makróparaméter mellékhatással veszélyes lehet.

```
int i = 2;
/* SQUARE(i++); */ // i többször értékelődhet
```

Általános szabály: ha nem kell preprocesszor-trükk, a típusos nyelvi eszköz gyakran könnyebben ellenőrizhető.

50. C API-tervezés: pointer + size

A biztonságosabb C API jellemzően nem csak pointert kap, hanem kapacitást/hosszt is.

```
bool format_user(char *dst, size_t cap,
                 const char *name, int id);

bool parse_packet(const uint8_t *data, size_t len,
                  struct Packet *out);
```

- Dokumentáld, lehet-e NULL.
- Dokumentáld, owned vagy borrowed-e a pointer.
- Mondd meg, ki és mikor free-z.
- Mondd meg, byte vagy elemszám a length.
- Stringnél mondd meg, kell-e NUL és a kapacitás tartalmazza-e a helyét.
- Hibánál legyen egyértelmű return code vagy státusz.

51. Python -> C mentális fordítótábla

Python	C megfelelő gondolat
x = 5	int x = 5; - típus és tárhely explicit
s = "abc"	const char *s="abc"; vagy char s[]="abc"; nem ugyanaz
len(s)	strlen(s) stringnél; sizeof más fogalom
a == b stringnél	strcmp(a,b)==0
list	tömb vagy heap-buffer + külön hossz/kapacitás
bytes	uint8_t*/unsigned char* + explicit length
None	NULL pointer bizonyos esetekben; nem univerzális objektum
exception	gyakran hibakód + errno/out-paraméter
object reference	pointer csak részben hasonló: nincs automatikus lifetime-védelem

Python referencia != C pointer. A Python referenciának nincs pointeraritmetikája, és az objektum élettartamát a runtime kezeli. C pointerrel neked kell követned az érvényességet.

52. Rust: miben más?

Téma	C	Rust
owned string	char* + konvenció	String
borrowed string	const char* + konvenció	&str;
slice	T* + külön hossz	&[T] / &mut; [T]
null	NULL	Option gyakori modell
heap	malloc/free	Box/Vec/String + RAII
hibakezelés	return code/errno	Result
lifetime	ember/API követi	borrow checker ellenőrzi

Rust fő nyeresége: sok ownership/lifetime/aliasing szabályt a fordító típus- és borrow rendszerben kényszerít ki. C-ben ezek nagy része dokumentáció és fegyelem.

String és UTF-8

Rust String és &str; érvényes UTF-8-at reprezentál. A C char* önmagában csak byte pointer: nem garantálja, hogy string, UTF-8 vagy akár NUL-lezárt.

53. Rust integer overflow és pointerek

Rustban a fix szélességű integer típusok explicit szélességűek: u8, i32, stb. Overflow viselkedés build módtól és művelettől függhet; explicit API-k: checked_*, wrapping_*, saturating_*.

Nyers pointer Rustban is létezik (*const T, *mut T), de dereferálása jellemzően unsafe környezetet igényel. A hétköznapi kód inkább referenciákat és slice-okat használ.

54. Gyakori hibák és javításuk

Hibás minta	Mi a baj?	Jobb gondolat
char *s; strcpy(s,"x");	s nem mutat allokált tárhelyre	char s[...]; vagy malloc
char s[3]="abc";	nincs hely NUL-nak	char s[4]="abc";
if (a==b) stringekre	címeket hasonlít	strcmp(a,b)==0
sizeof(p) stringhosszra	pointerméretet ad	strlen(p)
malloc(10) 10 inthez	csak 10 byte	malloc(10*sizeof *p)
return &local;	lejárt lifetime	heap vagy caller által adott storage
free(p); *p	use-after-free	free után ne dereferáld
strncpy = safe strcpy	nem garantál NUL-t	explicit kapacitáslogika

55. Mini program: biztonságosabb névbekérés

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

int main(void) {
    char name[32];

    printf("Name: ");
    if (!fgets(name, sizeof name, stdin))
        return 1;

    size_t len = strlen(name);
    if (len > 0 && name[len - 1] == '\n')
        name[--len] = '\0';

    if (strcmp(name, "admin") == 0)
        puts("special user");
    else
        printf("hello, %s\n", name);

    return 0;
}
```

Ebben együtt jelenik meg: fix buffer, kapacitásos input, newline/NUL, strlen, string-egyenlőség és %s.

56. Mini program: dinamikus integer tömb

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

int main(void) {
    size_t n = 100;

    if (n > SIZE_MAX / sizeof(int))
        return 1;

    int *a = malloc(n * sizeof *a);
    if (!a)
        return 1;

    for (size_t i = 0; i < n; ++i)
        a[i] = (int)i;

    printf("%d\n", a[42]);

    free(a);
    return 0;
}
```

57. Mini program: byte parser

```
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>

bool read_be16(const uint8_t *data, size_t len, uint16_t *out) {
    if (!data || !out || len < 2)
        return false;

    *out = ((uint16_t)data[0] << 8) |
           ((uint16_t)data[1]);

    return true;
}
```

Ez a minta jól mutatja a C API alapjait: const input, pointer + explicit length, out-paraméter, bool státusz, nincs pointer castos type punning.

58. Debug checklist

- Minden warningot olvass el és érts meg.
- Stringnél: biztosan van NUL a buffer határán belül?
- A kapacitás byte vagy elemszám?
- A +1 NUL bele van számolva?
- Pointer élő objektumra mutat?
- Nem történt free vagy scope-vége?
- Nincs double free?
- Nincs tömbhatár-túllépés?
- A printf formátum megfelel a tényleges típusnak?
- Nem lett signed/unsigned konverzióból meglepetés?
- A malloc méretszámítás nem csordul túl?
- Nincs hibás pointer cast / alignment probléma?

```
cc -g -O1 -Wall -Wextra -Wpedantic -Wconversion    -fsanitize=address,undefined program.c -o program
```

59. C string / pointer végső ellenőrzőlista

Kérdés	Miért fontos?
Hol van a memória?	stack/static/heap -> lifetime
Mekkora a kapacitás?	overflow elkerülés
Mennyi a hossz?	nem ugyanaz, mint kapacitás
Van NUL?	klasszikus string API feltétele
Ki birtokolja?	ki free-z?
Módosítható?	const/literál/storage kérdés
Hány elem érhető el pointertől?	pointer önmagában nem hordozza
UTF-8?	byte != karakter

Aranyszabály: pointer = cím jellegű érték. Nem tartalmaz automatikusan bufferhosszt, ownershipet, lifetime-ot vagy azt, hogy az adatok érvényes stringet alkotnak.

60. Egyoldalas gyors referencia

Feladat	Minta
integer értékadás	int x = 5;
integer összevetés	if (a == b)
string egyenlőség	strcmp(a,b) == 0
string hossz	strlen(s)
tömb elemszám helyben	sizeof a / sizeof a[0]
pointer címre	int *p = &x;
dereferálás	*p = 10;
pointerre pointer	int **pp = &p;
heap tömb	malloc(n * sizeof *p)
felszabadítás	free(p);
nyers másolás	memcpy(dst,src,n)
átfedő másolás	memmove(dst,src,n)
nyers összevetés	memcmp(a,b,n)
string formázás	snprintf(dst,cap,"%d",x)
szöveg -> long	strtol(text,&end;,10)
fix szélesség	uint16_t, int32_t
boolean	bool ok = true;
enum	enum State { ... };

Ha valami furcsa C-ben: írd le külön a típust, a tárhely méretét, a pointer által elérhető elemszámot, az élettartamot és az ownershipet. A legtöbb memória/string probléma ezek egyikének összekeveréséből indul.