C NYELV - BŐVÍTETT CHEAT SHEET

Pythonról érkezőknek - típusok, stringek, pointerek, memória, konverziók, UB, UTF-8 és Rust-kitekintés

Hogyan használd? Ez nem tankönyv, hanem kb. 3x-ra bővített gyors referencia. A sárga keretek fogalmat tisztáznak, a pirosak tipikus veszélyeket, a zöldek ajánlott mintát mutatnak. A példák főleg C11/C17 szemléletűek; ahol számít, C23-at külön jelzem.

Python -> C fő váltás: Pythonban egy változónév tipikusan objektumra hivatkozik, a runtime sok részletet kezel. C-ben a változó gyakran maga a fix típusú tárhely. Neked kell követned: típus, byte-méret, élettartam, tulajdonlás, kapacitás, pointer érvényesség.

1. C szabványok, fordító és a „mi garantált?” kérdés

C89/C90 a klasszikus alap. C99 nagy modernizálás: többek között <stdint.h>, // komment, deklarációk rugalmasabb helyen. C11 további nyelvi és concurrency elemeket hozott. C17 főleg korrekciós kiadás. C23 újabb kényelmi és nyelvi változásokat ad.

```
cc -std=c17 -Wall -Wextra -Wpedantic -Wconversion demo.c -o demo
# debugoláshoz, ha támogatott:
cc -std=c17 -g -fsanitize=address,undefined demo.c -o demo
```

A C szabvány szándékosan nem rögzít minden fizikai méretet. Például sizeof(char)==1 mindig igaz, de egy C byte nem elméletileg kötelezően 8 bit; CHAR_BIT adja meg. Mai általános gépeken jellemzően 8.

Három fontos kategória: defined = a szabvány meghatározza; implementation-defined = a fordító/platform választ, de dokumentálnia kell; undefined behavior (UB) = a szabvány nem ír elő viselkedést. UB után nem szabad arra építeni, hogy „nálam működött”.

2. Integer család - nem csak egyféle int létezik

Típus	Minimum / tipikus	Tipikus használat
signed char	legalább 8 bit	kis signed egész / nyers byte csak óvatosan
unsigned char	legalább 8 bit	nyers objektumreprezentáció byte-jaihoz különösen fontos
short	legalább 16; gyakran 16	kis egész
int	legalább 16; gyakran 32	alap egész
long	legalább 32; 32 vagy 64	platformfüggő széles egész
long long	legalább 64; gyakran 64	legalább 64 bites egész
unsigned változatok	azonos szélességi család	nemnegatív modulo aritmetika

signed int és int ugyanaz a típus. Hasonlóan long rövidítése a long int-nek, long long pedig long long int.

```
short a = 10;
int b = 20;
long c = 30L;
long long d = 40LL;

unsigned int u = 20U;
unsigned long ul = 30UL;
unsigned long long ull = 40ULL;
```

„not signed” = unsigned. Az unsigned nem azt jelenti, hogy „nagyobb int”, hanem más aritmetikai szabályokkal rendelkező nemnegatív egész típus.

Fix szélesség: int8_t, uint16_t, int32_t

<stdint.h> definiálja a fix szélességű típusokat, ha a platform képes pontosan ilyen típust biztosítani. A helyes név például uint16_t, nem uint_16t.

Név	Jelentés
int8_t / uint8_t	pontosan 8 bites signed / unsigned, ha létezik
int16_t / uint16_t	pontosan 16 bit
int32_t / uint32_t	pontosan 32 bit
int64_t / uint64_t	pontosan 64 bit
int_least32_t	legalább 32 bit, legkisebb megfelelő típus
int_fast32_t	legalább 32 bit, implementáció szerint gyors típus
intptr_t / uintptr_t	pointer érték reprezentálására képes integer, ha rendelkezésre áll

```
#include <stdint.h>
uint8_t flags = UINT8_C(0xA5);
uint16_t port = UINT16_C(443);
int32_t temperature = INT32_C(-1200);
```

3. Integer tartományok, signed/unsigned és túlcsordulás

N bites unsigned egész tipikus tartománya 0 .. 2^N-1. A C modern szabványaiban a signed reprezentáció kérdése fejlődött, de portábilis kódban továbbra is a szabványos limitekre építs: INT_MIN, INT_MAX, UINT_MAX stb.

```
#include <limits.h>
printf("%d %d\n", INT_MIN, INT_MAX);
printf("%u\n", UINT_MAX);
```

Unsigned wrap: UINT_MAX + 1u definiáltan 0-ra körbefordul. Signed overflow: pl. INT_MAX + 1 signed intként UB. Ez optimalizálásnál különösen fontos.

Signed és unsigned keveréskor a „usual arithmetic conversions” szabályai döntik el a közös típust. Emiatt egy negatív signed érték unsignedszá konvertálódhat.

```
int i = -1;
unsigned u = 1;
if (i < u) {
    // nem biztos, hogy azt jelenti, amit elsőre gondolsz
}
```

Python-különbség: Python int normál esetben tetszőleges pontosságú. C integer fix tartományú; a túlcsordulási szabály a konkrét típustól függ.

4. Literálok, printf és scanf formátumok

Integer literál: 123, oktális 0173, hex 0x7B; C23-ban bináris 0b1111011. Suffixek: U, L, LL.

Érték	Példa
int	42
unsigned	42U
long	42L
unsigned long	42UL
long long	42LL
unsigned long long	42ULL

printf-nél a formátum és a tényleges argumentumtípus egyezése kritikus. %d int, %u unsigned int, %ld long, %lld long long, %zu size_t.

```
size_t n = 10;
long x = 123;
printf("n=%zu x=%ld\n", n, x);

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

Miért PRIu32? A uint32_t mögötti konkrét alap integer típus platformfüggő lehet. Az <inttypes.h> makrói portábilis formátumot adnak.

5. 32 bit vs 64 bit - adatmodellek

Modell	short	int	long	long long	pointer
ILP32	16	32	32	64	32
LP64	16	32	64	64	64
LLP64	16	32	32	64	64

Linux/macOS 64 bit tipikusan LP64, Windows 64 bit tipikusan LLP64. Ezért hálózati protokollhoz, fájlformátumhoz vagy bináris struktúrához ne használj vakon long-ot „64 bites szám” jelentésben.

Pointer méret != int méret. 64 bites gépen egy pointer tipikusan 8 byte, miközben int továbbra is 4 byte.

6. float, double, long double

Típus	Gyakori méret	Tipikus pontosság	Literál
float	32 bit	~6-9 decimális számjegy	1.0f
double	64 bit	~15-17	1.0
long double	64/80/96/128	implementációfüggő	1.0L

Lebegőpontos számok bináris közelítések. Például a 0.1 általában nem reprezentálható pontosan. Ez nem C-hiba, hanem az IEEE-754 jellegű bináris lebegőpont természetes következménye.

```
double a = 0.1 + 0.2;
double b = 0.3;
// a == b lehet hamis
if (fabs(a - b) < 1e-12) { /* közel azonos */ }
```

Pénzhez: sok esetben jobb egész egységet használni, pl. centet int64_t-ban, vagy megfelelő decimális könyvtárat. A bináris float nem automatikusan pénzügyi decimális típus.

7. Automatikus konverziók és cast

C sok implicit konverziót végez: integer promotion, signed/unsigned közös típus, integer -> float, kisebb -> nagyobb típus. Ez kényelmes, de hibaforrás.

```
uint8_t a = 250, b = 10;
int sum = a + b;       // a és b tipikusan int-re promotálódik

double d = 3.9;
int i = (int)d;        // 3, ha reprezentálható

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

Egy explicit cast azt mondja a fordítónak, hogy szándékosan kéred a konverziót. Nem ellenőrzi, hogy üzletileg helyes-e.

Pointer cast különösen veszélyes. (int*)some_char_pointer nem alakítja át a mögöttes byte-okat intté. Csak más típusú pointerként kezdi értelmezni ugyanazt a címet, ami alignment/aliasing/méret miatt UB lehet.

8. const részletesebben

Deklaráció	Olvasd így
const int x	x értékét ezen a néven át nem módosíthatod
const char *p	p módosítható, de *p nem módosítható p-n keresztül
char * const p	p nem állítható más címre, de *p módosítható
const char * const p	sem p, sem *p nem módosítható ezen a hozzáférésen át

```
char buf[] = "abc";
const char *p = buf;
p++;              // OK
// *p = 'X';      // nem OK p-n keresztül

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

const nem feltétlen fizikai read-only memória. Azt fejezi ki, hogy az adott hozzáférési útvonalon nem módosítható. String-literálnál viszont a módosítás eleve UB.

9. static három gyakori jelentése

Lokális static: a változó élettartama a teljes programfutás, de a neve csak a blokkban látszik. Fájlszintű static változó/függvény: internal linkage, más fordítási egységből nem érhető el ezen a néven.

```
void counter(void) {
    static unsigned calls = 0;
    calls++;
}

static int internal_state;
static void helper(void) { }
```

A static tárolási idejű objektumok programindításkor nullára inicializálódnak, ha nincs explicit inicializálójuk.

Nem thread-safe automatikusan: egy lokális static közös állapotot jelenthet több hívás között; többszálú kódban szinkronizációs kérdéseket vethet fel.

10. Tömbök: valódi tömb vs pointer

```
int a[4] = {10, 20, 30, 40};
size_t count = sizeof a / sizeof a[0];

int *p = a;      // legtöbb kifejezésben a tömb első elemére mutató pointerré alakul
printf("%d", p[2]); // 30
printf("%d", *(p + 2)); // 30
```

A p[i] definíció szerint lényegében *(p+i). A pointeraritmetika elemekben lép: p+1 a következő int-re mutat, nem a következő byte-ra.

sizeof csapda: valódi tömbnél sizeof a az egész tömb byte-mérete. Függvényparaméterben int a[] már pointerként viselkedik, ezért ott sizeof a pointerméretet ad.

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

A C string olyan char sorozat, amelynek végét egy 0 értékű byte, a NUL jelzi. A string könyvtári függvények általában addig olvasnak, amíg ezt meg nem találják.

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

Jelölés	Mi ez?
'\0'	NUL karakter, numerikus értéke 0; C string lezáró
NULL	null pointer konstans/makró pointerekhez
'\n'	newline / line feed karakter
'\r'	carriage return
"\r\n"	CRLF, klasszikus DOS/Windows szöveges sorvég

A string nem azért ér véget, mert elfogyott a tömb. A stringfüggvények NUL-t keresnek. Ha egy char bufferben nincs NUL a hozzáférhető tartományban, a strlen/printf("%s") túlolvasást okozhat.

12. char*, char[], char[10] - részletes minták

```
char a[] = "hello";        // 6 byte: h e l l o NUL
char b[10] = "hello";      // 10 byte kapacitás
const char *c = "hello";   // pointer literálra

a[0] = 'H';                // OK
b[5] = '!';                // ezzel felülírod az eredeti NUL-t
b[6] = '\0';               // új lezárás
// c[0] = 'H';              // string-literál módosítása UB
```

Kifejezés	Mit tudsz róla?
char *p	egy cím; a kapacitás ismeretlen, hacsak külön nem tárolod
char a[]	a fordító az inicializálóból számolja az elemszámot
char a[10]	pontosan 10 char tárhely
const char *p	olvasási hozzáférésre alkalmas pointer

Egy 8 byte hosszú ASCII szöveghez 9 byte kell NUL-lal. Ha a buffer char s[8], akkor csak 7 byte hosszú NUL-lezárt string fér bele.

Kapacitás vs hossz: char b[10]="hi" kapacitása 10 byte, aktuális stringhossza 2. Ezt a két fogalmat mindig külön kezeld.

13. String literál, tömb és pointer memóriaképe

char a[]="abc" létrehoz egy saját 4 elemű tömböt, amelybe a literál tartalma bemásolódik. const char *p="abc" csak egy pointert tárol a literál karaktereire.

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

printf("%zu\n", sizeof a); // 4
printf("%zu\n", sizeof p); // pointer mérete, pl. 8
printf("%zu\n", strlen(p)); // 3
```

sizeof != strlen. sizeof típus/tárhely méretét adja byte-ban, és tömbnél a NUL is benne van, ha része a tömbnek. strlen futásidőben NUL-ig számol byte-okat.

14. =, ==, strcmp, strncmp

```
int x = 5;          // = értékadás
if (x == 5) { }    // == integer értékegyenlőség

char a[] = "alma";
char b[] = "alma";

if (a == b) { }             // címek összevetése, nem tartalom
if (strcmp(a, b) == 0) { }  // tartalmi string-egyenlőség
```

strcmp lexikografikus byte-összehasonlítást végez. 0 = azonos; negatív = első kisebb; pozitív = első nagyobb. strncmp(a,b,n) legfeljebb n karaktert/byte-ot vizsgál, ezért prefixvizsgálathoz jó lehet, de nem mindig teljes string-egyenlőség.

Gyakori bug: if (x = 5) értékadást végez, majd az eredményt feltételként értékeli. Warningokkal a fordító sok ilyen hibát jelez.

15. strcpy miért veszélyes? strncpy miért furcsa?

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

strcpy addig másol, amíg NUL-t nem talál, és nem kap célkapacitást. Emiatt önmagában nem tudja megakadályozni a túlcsordulást.

```
char dst[5];
strncpy(dst, "elephant", sizeof dst);
// dst valószínűleg: e l e p h
// NINCS garantált NUL, mert a forrás nem fért bele
```

strncpy eredetileg fix szélességű mezők kezelésére hasznos minta. Ha a forrás rövidebb n-nél, nullákkal tölti ki a teljes maradékot; ha legalább n hosszú, nem ír lezáró NUL-t.

Ezért „strncpy biztonságos strcpy” rossz mentális modell. Mindig a kívánt szemantikából indulj ki: hibázzon, ha nem fér bele? Csonkoljon? Pontos byte-másolat kell?

16. Ajánlott stringmásolási minták

A) Nem engedünk csonkolást

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

B) Formázott írás kapacitással

```
char dst[16];
int n = snprintf(dst, sizeof dst, "%s", src);
if (n < 0) {
    // formázási hiba
} else if ((size_t)n >= sizeof dst) {
    // csonkolás történt
}
```

A snprintf jó általános eszköz formázott szöveghez, de a visszatérési értéket értelmezni kell. Ha a kapacitás > 0, megfelelő használat mellett NUL-lezárást ad.

C) Nyers byte-másolás

```
memcpy(dst, src, n);   // csak ha legalább n byte érvényes mindkét oldalon
memmove(dst, src, n);  // átfedés esetén is használható
```

17. String beolvasás: scanf helyett kapacitástudatosan

scanf("%s", buf) szélességkorlát nélkül veszélyes lehet. Sorbeolvasásra gyakran egyszerűbb a fgets.

```
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';
    }
}
```

fgets megtarthatja a newline-t. Ha a sor belefér és volt sorvég, a \n bekerülhet a bufferbe a NUL elé. Ezért string-egyenlőségnél gyakran előbb el kell távolítani.

18. Szöveg -> szám: atoi helyett strtol

atoi hibajelzése gyenge. Robusztusabb a strtol/strtoul/strtod, mert megmondható, hol állt meg a feldolgozás, és errno-val tartományhiba is kezelhető.

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

if (end == text) { /* nem volt szám */ }
else if (*end != '\0') { /* maradt szemét */ }
else if (errno == ERANGE) { /* túl kicsi/nagy */ }
else { /* v használható */ }
```

Python párhuzam: Pythonban int("123") exceptiont dob hibánál. C-ben a hibakezelés gyakran visszatérési érték + out-pointer + errno kombináció.

19. Pointer alapok - &, *, cím és érték

```
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);  // p változó saját címe
```

&x = „x címe”. *p kifejezésben = „a p által mutatott objektum”. Deklarációban az int *p azt mondja: p típusa pointer int-re.

A csillagnak két közeli szerepe van: deklarációban pointertípust jelöl, kifejezésben dereferál. A kontextus dönti el.

20. Pointerre mutató pointer: int **

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

**pp = 77;      // x = 77
*p = 88;        // x = 88
p = NULL;       // pp továbbra is p változó címére mutat
*pp = &x;       // p ismét &x
```

pp nem közvetlenül x-re mutat, hanem arra a változóra, amely x címét tárolja. Ez hasznos, ha egy függvénynek magát a pointerváltozót kell módosítania.

```
bool allocate_int(int **out) {
    *out = malloc(sizeof **out);
    if (*out == NULL) return false;
    **out = 123;
    return true;
}

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

21. NULL, dangling pointer, ownership

NULL azt jelzi, hogy egy pointer nem mutat érvényes objektumra. De nem minden érvénytelen pointer NULL: felszabadítás után a régi cím dangling pointer.

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

*p = 10;
free(p);
// *p = 20;  // use-after-free: UB
p = NULL;     // segíthet elkerülni ugyanennek a változónak a véletlen újrahasználatát
```

free(NULL) megengedett. Viszont ugyanazt az allokációt kétszer felszabadítani double free és UB. Több alias pointer esetén az egyik NULL-ra állítása nem nullázza a többit.

22. Pointeraritmetika és one-past-the-end

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

p++;          // most a[1]-re mutat
int x = p[1]; // a[2] -> 30
```

Pointeraritmetika csak megfelelő objektum/tömb határain belül értelmes. Képezhető a tömb utáni egy pozícióra mutató pointer („one past”), de azt nem szabad dereferálni.

```
int *begin = a;
int *end = a + 4;
for (int *it = begin; it != end; ++it) {
    printf("%d\n", *it);
}
```

Nem általános cím-matematika. Tetszőleges, egymástól független objektumok pointereinek kivonása/rendezése nem olyan, mint Pythonban számokat hasonlítani.

23. uint8_t érték int*-en keresztül - helyes és helytelen

```
uint8_t small = 200;
int x = 0;
int *p = &x;
*p = small;       // értékkonverzió, helyes: x = 200
```

Itt a small értékét olvassuk ki, intté konvertáljuk, majd egy valódi int objektumba írjuk.

```
uint8_t small = 200;
uint8_t *q = &small;
int *bad = (int*)q;
// printf("%d", *bad);  // lehet hibás alignment, túl kevés tárhely, aliasing UB
```

A cast nem varázsol 1 byte-ból int objektumot. Ha byte-okból kell 32 bites számot dekódolni, explicit összerakás vagy memcpy + endianness-kezelés kell.

24. Endianness - amikor byte-okból integer lesz

A többbyte-os integer memóriabeli byte-sorrendje lehet little-endian vagy big-endian. Hálózati protokollok és fájlformátumok gyakran rögzített sorrendet használnak.

```
uint8_t b[2] = {0x12, 0x34};
uint16_t value = ((uint16_t)b[0] << 8) | b[1];
// value = 0x1234, a host endiannessétől függetlenül
```

Ne castold vakon: *(uint16_t*)b nemcsak endianness miatt problémás, hanem alignment és aliasing miatt is lehet helytelen.

25. malloc, calloc, realloc, free

Függvény	Lényeg
malloc(n)	n byte inicializálatlan dinamikus tárhely
calloc(count,size)	count*size byte, byte-szinten nullázott tárhely
realloc(p,newsize)	allokáció méretének változtatása; elmozdulhat
free(p)	korábbi dinamikus allokáció felszabadítása

```
size_t n = 100;
if (n > SIZE_MAX / sizeof(int)) {
    // szorzás túlcsordulna
}
int *a = malloc(n * sizeof *a);
if (!a) { /* hiba */ }

free(a);
```

Miért sizeof *a? Nem kell kézzel ismételni a típust. Ha később int*-ről más pointertípusra változtatod a deklarációt, a méretszámítás együtt változik.

26. realloc biztonságos minta

```
size_t new_count = 200;
int *tmp = realloc(a, new_count * sizeof *a);
if (tmp == NULL) {
    // a még mindig a régi allokációra mutat
} else {
    a = tmp;
}
```

Ha közvetlenül a = realloc(a,...)-t írsz, és a realloc hibázik, elveszítheted a régi pointert, így memóriaszivárgás lesz.

realloc sajátosság: siker esetén a régi pointert többé ne használd; az adat új címre kerülhetett.

27. Stack, static storage, heap - élettartam

Hely / tárolási idő	Példa	Meddig él?
automatikus lokális	int x; egy függvényben	blokk/függvény végéig
static storage	static int x; vagy globális	program teljes futásáig
heap/dinamikus	malloc	free-ig

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

int *ok(void) {
    int *p = malloc(sizeof *p);
    if (p) *p = 5;
    return p;       // callernek kell free
}
```

28. sizeof részletesen

sizeof(T) egy típus byte-méretét, sizeof expr az expression típusának méretét adja. A legtöbb esetben az expressiont nem értékeli ki.

```
int a[10];
printf("%zu\n", sizeof(int));
printf("%zu\n", sizeof a);       // 10 * sizeof(int)
printf("%zu\n", sizeof a[0]);    // sizeof(int)
printf("%zu\n", sizeof a / sizeof a[0]); // 10
```

VLA kivétel: változó hosszúságú tömbök környezetében a sizeof értékelési részletei eltérhetnek. Kezdő szinten a lényeg: normál fix típusnál sizeof nem „lefuttatja” az argumentumot.

29. Struct, padding és sizeof

```
struct Point {
    char tag;
    int x;
    int y;
};

printf("%zu", sizeof(struct Point));
```

A struct mérete lehet nagyobb a mezők puszta összegénél, mert a fordító alignment miatt padding byte-okat illeszthet be.

Bináris fájl/protokoll: ne feltételezd, hogy egy struct memóriaképe automatikusan hordozható más platformra vagy fordítóra. Padding, endianness és típusméretek eltérhetnek.

30. String allokáció és ownership

```
char *dup_string(const char *src) {
    size_t len = strlen(src);
    if (len == SIZE_MAX) return NULL; // elméleti védelem
    char *dst = malloc(len + 1);
    if (!dst) return NULL;
    memcpy(dst, src, len + 1);
    return dst;
}

char *s = dup_string("hello");
if (s) {
    puts(s);
    free(s);
}
```

Itt a visszaadott pointer egy új, owned heap-allokációra mutat. Az API dokumentációjának meg kell mondania, ki felel a free-ért.

C-ben ownership konvenció. A típusrendszer önmagában nem jelöli, hogy egy char* owned, borrowed, statikus vagy felszabadítandó. Ezt névvel, dokumentációval és API-tervezéssel kell világossá tenni.

31. Buffer + length: sokszor jobb, mint csak C string

Bináris adathoz vagy beágyazott NUL-t tartalmazó adathoz a klasszikus C string nem elég. Használj pointer + explicit hossz párost.

```
struct ByteSlice {
    const unsigned char *data;
    size_t len;
};

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

Ez a gondolkodás közelebb áll Rust slice-hoz vagy Python bytes-hoz: a tartomány végét nem egy sentinel byte jelzi, hanem külön hossz.

32. memcmp vs strcmp

strcmp NUL-lezárt stringet hasonlít. memcmp(a,b,n) pontosan n byte-ot hasonlít, ezért bináris adathoz vagy ismert hosszú byte-szeletekhez használható.

```
unsigned char a[3] = {0, 1, 2};
unsigned char b[3] = {0, 1, 2};
if (memcmp(a, b, 3) == 0) { /* azonos 3 byte */ }
```

memcmp nem Unicode/string szemantika. Nyers objektumstruktúrák összevetésére sem mindig jó, mert padding byte-ok lehetnek nem meghatározott értékűek.

33. UTF-8 C-ben - byte, kódpont, grapheme

Három külön fogalom: byte = tárolási egység; Unicode kódpont = pl. U+00E9; grapheme cluster = amit a felhasználó egy karakternek érzékelhet. UTF-8-ban egy kódpont 1-4 byte.

```
const char *s = "árvíz";
printf("%zu", strlen(s)); // UTF-8-ban byte-szám, nem feltétlen 5
```

ASCII tartományban 1 karakter = 1 byte, ezért a különbség sokáig rejtve maradhat. Magyar ékezetes betűk UTF-8-ban tipikusan több byte-osak.

Indexelés: s[1] a második byte, nem garantáltan a második Unicode karakter. Tetszőleges bytehatáron vágni érvénytelen UTF-8-at hozhat létre.

34. UTF-8 összehasonlítás

Ha két stringet pontosan azonos UTF-8 byte-sorozatként akarsz összevetni, strcmp megfelel. Ha emberi szövegként akarsz összehasonlítani, több kérdés jön elő: Unicode normalizáció, case folding, locale, rendezési szabályok.

Példa: az „é” előállhat egyetlen U+00E9 kódpontként vagy „e” + combining acute accent sorozatként. Vizuálisan hasonló/azonos lehet, de byte-szinten más.

Komoly Unicode feldolgozáshoz célszerű Unicode-könyvtárat használni (pl. ICU), vagy olyan magasabb szintű környezetet, ahol a szükséges normalizáció/collation API rendelkezésre áll.

35. Locale és strcmp

strcmp nem nyelvi ábécésorrendet implementál, hanem byte-alapú lexikografikus összehasonlítást. A C locale-függő könyvtári eszközöket is kínál, például strcoll, de Unicode/modern nemzetközi szöveghez platform- és könyvtárválasztás szükséges.

„alma” < „Árvíz” emberi rendezés szerint? Ez locale- és szabályfüggő kérdés; nem oldható meg megbízhatóan pusztán strcmp-pal.

36. Függvényparaméterek: érték szerinti átadás

C-ben a paraméterek érték szerint adódnak át. Pointer átadásakor is a pointerérték másolata kerül a függvénybe.

```
void set_bad(int x) {
    x = 10;          // caller változója nem változik
}

void set_ok(int *x) {
    *x = 10;         // a caller objektumát módosítjuk
}

int n = 1;
set_ok(&n);
```

Python párhuzam csak óvatosan: Python objektumreferenciás modellje nem azonos a C pointerrel. C-ben explicit cím, dereferálás, objektumélettartam és pointeraritmetika van.

37. Out-paraméter és pointer-to-pointer

```
bool parse_number(const char *text, long *out) {
    char *end;
    errno = 0;
    long v = strtol(text, &end, 10);
    if (errno || end == text || *end != '\0') return false;
    *out = v;
    return true;
}

long value;
if (parse_number("123", &value)) { /* value == 123 */ }
```

C API-kban gyakori minta: a függvény visszatérési értéke siker/hiba, a tényleges eredmény pointeres out-paraméterbe kerül.

38. Undefined behavior - rövid 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	char *p="x"; p[0]='Y'
hibás shift	túl nagy vagy negatív shift count
uninitialized read	int x; printf("%d",x)

UB nem azt jelenti, hogy „véletlenszerű értéket kapsz”. A fordító optimalizációja feltételezheti, hogy UB nem történik, és ennek alapján teljes kódrészeket alakíthat át.

Sanitizerek: fejlesztéskor AddressSanitizer (ASan) sok memóriahibát, UndefinedBehaviorSanitizer (UBSan) sok UB-t képes futásidőben felismerni. Nem bizonyítják a hibamentességet, de rendkívül hasznosak.

39. Inicializálás

```
int x = 0;
int a[10] = {0};          // minden elem 0
struct Point p = {0};     // mezők nullázott értékre inicializálva

int *heap = calloc(10, sizeof *heap); // byte-szinten nullázott tárhely
```

Automatikus lokális változó explicit inicializálás nélkül indeterminált értékkel indulhat; olvasása bizonyos esetekben UB. Static storage objektumok implicit null-inicializálást kapnak.

calloc és pointerek: a calloc byte-szinten nulláz. Portábilis elméleti szinten ne általánosítsd túl, hogy minden típus „minden-bites-zero” reprezentációja minden esetben a nyelvi nulla/null érték; hétköznapi modern platformokon ez gyakran működik, de a fogalmak különböznek.

40. 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) { /* be van állítva */ }
flags &= ~FLAG_WRITE;    // törlés
flags |= FLAG_EXEC;      // beállítás
```

& bitwise AND, | OR, ^ XOR, ~ NOT, <</>> shift. Ne keverd a logikai &&, ||, ! operátorokkal.

Bitműveleteknél unsigned típus gyakran könnyebben kezelhető, mert a signed shift és reprezentáció több finomságot hordoz.

41. bool C-ben

```
#include <stdbool.h>
bool ok = true;
if (!ok) {
    // ...
}
```

C99 óta <stdbool.h> kényelmes bool, true, false neveket ad a nyelvi _Bool köré. Feltételben a 0 hamis, nemzero igaz.

strcmp visszatérési értéke ezért csapda: azonos stringeknél strcmp(a,b) 0, tehát feltételben hamis. Egyenlőséghez írd ki expliciten: strcmp(a,b)==0.

42. Makrók vs const

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

A preprocessor makró szöveges helyettesítés. A const valódi típusos C objektum. Sok egyszerű konstansnál a típusos megoldás könnyebben ellenőrizhető, de fordítási idejű konstans-követelmények és C-verziók miatt a megfelelő eszköz kontextusfüggő.

```
#define SQUARE(x) ((x) * (x))
// SQUARE(i++) veszélyes: i többször értékelődhet
```

Makróparaméter mellékhatással: a preprocessor nem függvény. Inline függvény sok esetben biztonságosabb és típusosabb.

43. enum

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

enum State state = STATE_IDLE;
```

Az enum olvasható neveket ad egész konstansoknak. A pontos reprezentáció és méret szabvány/platform részleteitől függhet; külső bináris formátumban ne támaszkodj vakon rá.

44. struct és pointer structra

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

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

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

p->field a pointeren keresztüli mezőelérés rövidítése. Structot érték szerint is lehet másolni/átadni, de nagy objektumnál pointeres API gyakori.

45. Function pointer - rövid kitekintés

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

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

A function pointer függvény címét tárolja. Callback API-kban gyakori. A deklaráció elsőre nehéz: int (*op)(int,int) = op pointer olyan függvényre, amely két intet kap és intet ad vissza.

46. C string API mini referencia

Függvény	Feladat / figyelmeztetés
strlen	NUL előtti bytehossz; O(n), érvényes lezárt string kell
strcmp	teljes string byte-alapú összevetése
strncmp	legfeljebb n byte/karakter összevetése
strchr	karakter első előfordulása
strstr	substring keresése
strcpy	kapacitást nem tud; kerülendő ellenőrzés nélkül
strncpy	furcsa padding/NUL szemantika
strcat	kapacitást nem tud; veszélyes ellenőrzés nélkül
snprintf	kapacitást kapó formázott írás

47. „Mekkora buffer kell?” receptek

```
// "hello" másolata
size_t need = strlen(src) + 1; // + NUL
char *p = malloc(need);

// n darab int
if (n <= SIZE_MAX / sizeof(int))
    int *a = malloc(n * sizeof *a);

// formázott szöveg méretének lekérdezése
```

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

Mindig tedd fel: az egység byte vagy elemszám? Benne van-e a NUL? Lehet-e túlcsordulás a szorzásban/összeadásban?

48. Rust: miért érződik másnak ugyanaz a probléma?

Rust célja nem az, hogy „C szintaxis biztonsági extrákkal” legyen. A típus- és ownership modellje alapból megpróbálja reprezentálni azt, amit C-ben konvencióként kell fejben tartani.

C fogalom	Rust közelítő pár
char * owned heap string	String
const char * + ismert UTF-8 szelet	&str;
T * + elemszám	&[T] vagy &mut; [T]
NULL-lehetséges pointer	Option<&T;>, Option> stb.
malloc/free	Box, Vec, String + RAII
hibakód	Result

Rust slice: a referencia mellett a hossz is a típus/runtime reprezentáció része. C-ben a T* önmagában nem mondja meg, hány elem érhető el.

49. Rust ownership vs C ownership

C-ben egy API dokumentációja mondhatja: „a caller owns the returned pointer and must free it”. Rustban az owned érték mozgatása és a borrow szabályok a fordító által ellenőrzött modell részei.

```
/* C */
char *make_name(void); // dokumentációból kell tudni, ki free-zze

/* Rust gondolat:
fn make_name() -> String
*/
```

Safe Rustban a use-after-free, sok dangling referencia és sok adatverseny már fordításkor kizárható. Ez nem jelenti, hogy Rustban nincs bug, csak a lehetséges hibák jelentős osztályát más réteg kezeli.

50. Rust String és UTF-8

Rust String és &str garantáltan érvényes UTF-8. Emiatt a s[0] jellegű byte-indexelés stringre nincs úgy engedve, mint C-ben: egy byteindex nem feltétlen karakterhatár.

C-ben: a char* csak byte-okra mutat; a típus nem garantálja, hogy UTF-8, NUL-lezárt, vagy akár szöveg. Rustban: a &str már erősebb invariánst hordoz.

51. Python, C és Rust - mentális térkép

Kérdés	Python	C	Rust
Mi egy string?	Unicode objektum	konvenció szerint NUL-lezárt char byte-ok	UTF-8 String/&str;
Ki kezeli a memóriát?	runtime	programozó/API konvenció	ownership + RAII
Hossz hol van?	objektumban	gyakran külön vagy NUL-ból számolva	slice/String reprezentációban
Index bounds?	ellenőrzött	tipikusan nincs	safe indexelés ellenőrzött
Null?	None	NULL pointer	Option
Integer	nagy pontosságú	fix típus/tartomány	fix típus, explicit overflow API-k is

52. Jó C API tervezési minták

A biztonságot sokszor az API formája dönti el. Jobb write(dst, cap, src), mint write(dst, src). Jobb pointer + hossz párost adni bináris adathoz. Egyértelműen dokumentáld a NULL engedélyezését és ownershipet.

```
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);
```

Hasznos kérdések API olvasásakor: Ki allokál? Ki free-z? Lehet NULL? Meddig érvényes a visszaadott pointer? A függvény módosítja a buffert? Mekkora buffer kell? Hogyan jelzi a hibát?

53. Debug checklist memóriahibához

Ha egy program néha összeomlik vagy „furcsa stringet” ír ki, vizsgáld sorrendben: 1) van-e NUL; 2) helyes-e a buffer kapacitása; 3) nincs-e indexhatár-túllépés; 4) él-e még a memória; 5) nem történt-e double free; 6) helyes-e a printf formátum; 7) nem lett-e pointertípus hibásan castolva; 8) nincs-e integer overflow a méretszámításban.

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

A fordító warningokat ne kezeld zajként. Tanulási projektnél érdemes magas warningszintet használni, és megérteni minden figyelmeztetést.

54. Gyakori kezdőhibák - gyors javítás

Hiba	Jobb gondolat
char *s; strcpy(s,"x");	s nem mutat allokált bufferre; előbb tárhely kell
char s[3]="abc";	nincs hely NUL-nak; stringként nem biztonságos
if (a==b) stringekre	strcmp(a,b)==0
malloc(10) int tömbhöz	malloc(10 * sizeof *p)
return &local;	lokális objektum címe érvénytelenné válik
free(p); use(p);	free után a pointee élettartama véget ért
sizeof(p) a string hosszára	strlen(p), ha érvényes C string
strncpy = safe strcpy	nem; szemantikát és NUL-t külön kezeld

55. Mini példaprogram: biztonságosabb névbekérés

```
#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 && name[len - 1] == '\n')
        name[--len] = '\0';

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

    return 0;
}
```

Ebben egyszerre látszik a fix buffer, kapacitásos beolvasás, newline eltávolítás, string-egyenlőség és %s használat.

56. Mini példaprogram: dinamikus int 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. Végső „C-ben ezt mindig tudd” lista

Integer: pontos típus, signedness, tartomány, konverzió. Tömb/string: kapacitás, aktuális hossz, NUL. Pointer: mire mutat, hány elem érhető el, él-e még, ki birtokolja. Heap: ki allokálta, ki szabadítja fel, lehet-e realloc. Szöveg: ASCII vagy UTF-8,

byte vagy karakter a mértékegység. API: hibajelzés, NULL-szabály, ownership.

Ha csak egy mondat marad meg: C-ben a pointer önmagában csak egy cím jellegű érték - nem hordoz automatikusan típustól független futásidejű információt a buffer hosszáról, élettartamáról, ownershipéről vagy arról, hogy a byte-ok érvényes stringet alkotnak-e. Ezeket a program struktúrájának kell helyesen fenntartania.