Kvíz

- http://www.menti.com
- mindenki a saját azonosítójával (bbbbnnnn) lépjen be

! aki **nem** az azonosítóját használja, annak nem fogjuk tudni beírni a pontokat a kvízre

1251 9558

Struktúrák

KURZUS

Felhasználó által definiált típusok

- a **typedef** kulcsszó és az ismert adattípusok segítségével új típusokat tudunk definiálni

```
typedef ismertTípus újTípus;
```

példa `typedef long unsigned int size_t;`

példa `typedef int* pointerInt_re;`

1251 9558 kvíz 1. kérdés

Példa

typedef.c

```
> az 1. es 2. sorok elemeinek
szorzatosszege: 278
```

```c
#include <stdio.h>

typedef int egesz;
typedef egesz vektor[4];
typedef vektor matrix[5];

egesz szorzatOsszege(vektor x, vektor y)
{
    egesz i, sum = 0;
    for(i = 0; i < 4; i++)
    {
        sum += x[i] * y[i];
    }
    return sum;
}

int main()
{
    matrix M = {{1, 2, 3, 4}, {5, 6, 7, 8}, \
       {9, 10, 11, 12}, {13, 14, 15, 16}, {17, 18, 19, 20}};
    egesz osszeg = szorzatOsszege(M[1], M[2]);
    printf("> az 1. es 2. sorok elemeinek szorzatosszege: %d\n", osszeg);

    return 0;
}
```

egy egész típus

az új **egesz** típust használva létrehozunk egy **vektor** típust

a **vektor** típus segítségével létrehozunk egy **matrix** típust

az új típusokat használva deklaráljuk a változókat

Struktúrák

- több változó csoportosítása

tömb

↓

azonos típusú elemek gyűjteménye

struktúra

↓

különböző típusú elemek gyűjteménye

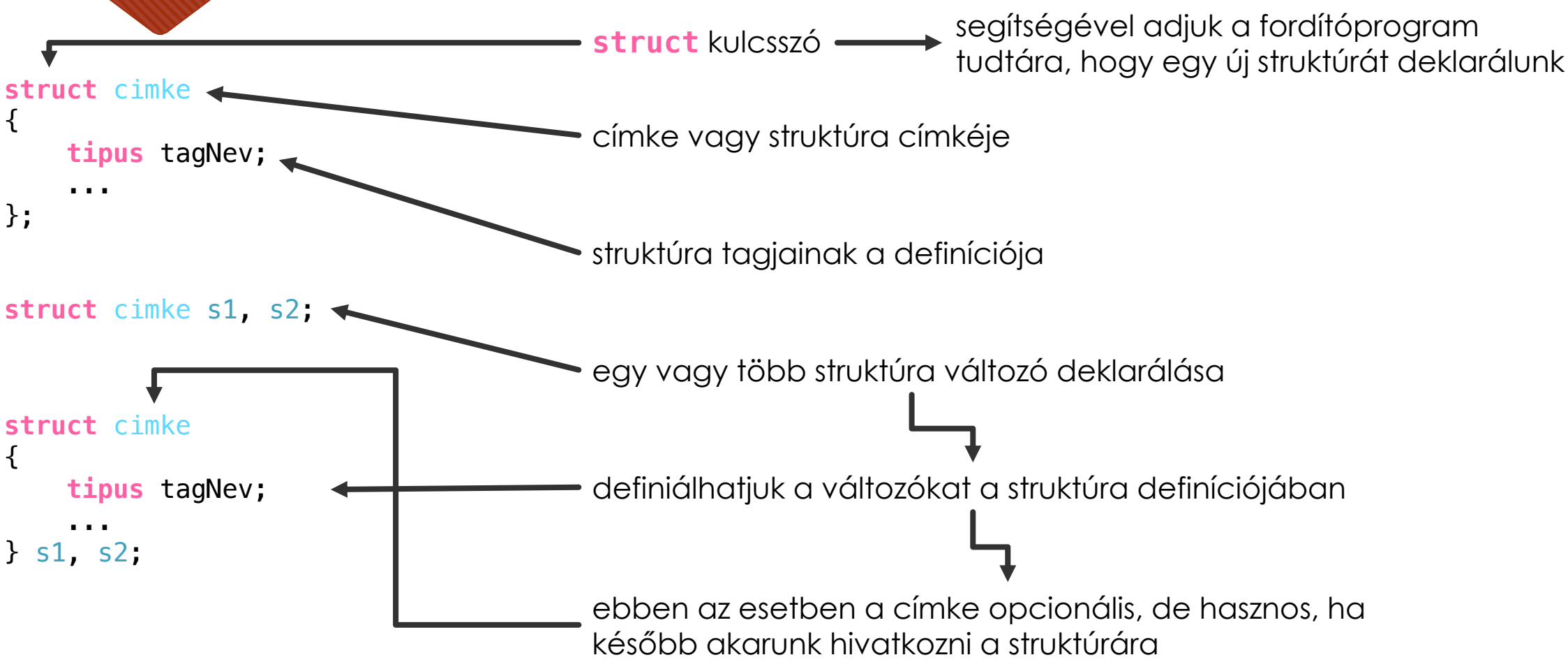
↓

lehetővé teszi összetett típusok létrehozását

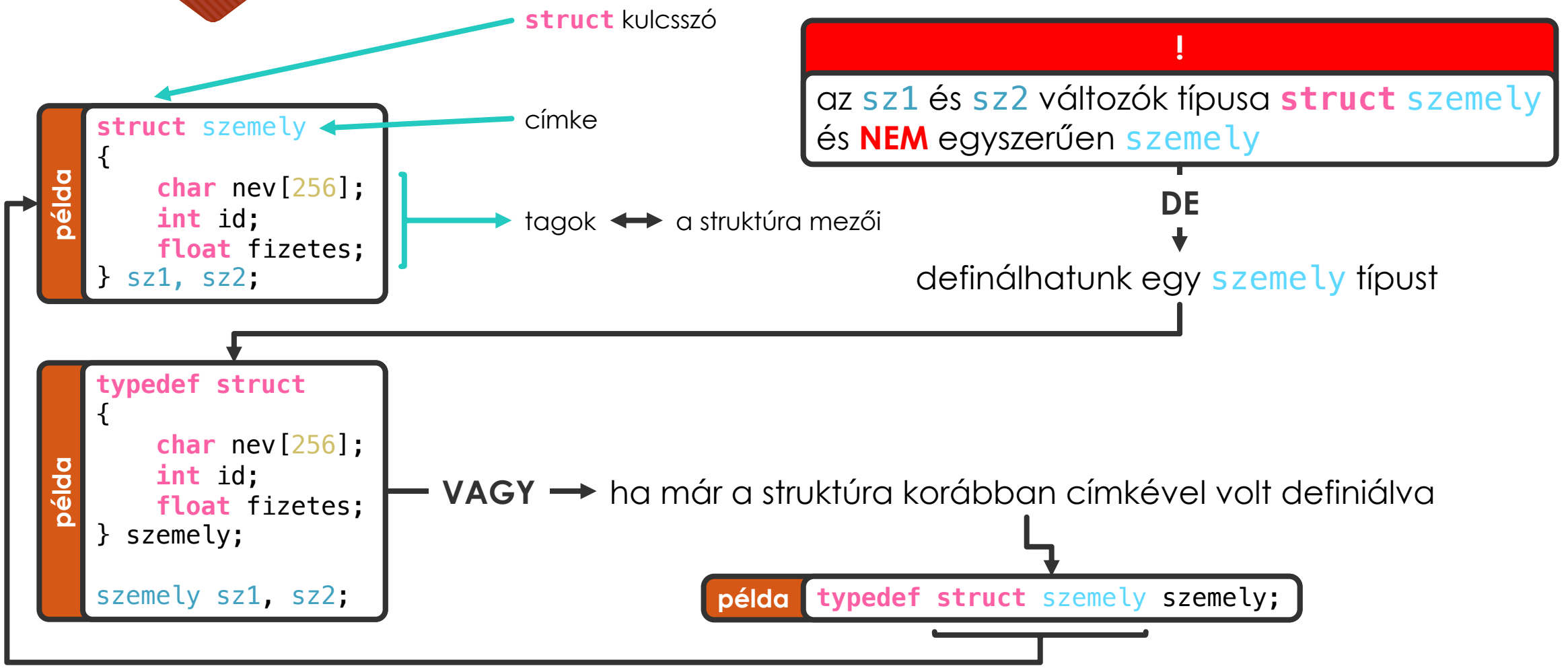
↓

összefoglal különböző alap adattípusokat

Struktúrák


struct kulcsszó
segítségével adjuk a fordítóprogram tudtára, hogy egy új struktúrát deklarálunk
struct cimke
{
tipus tagNev;
...
};
címke vagy struktúra címkéje
struktúra tagjainak a definíciója
struct cimke s1, s2;
egy vagy több struktúra változó deklarálása
struct cimke
{
tipus tagNev;
...
} s1, s2;
definiálhatjuk a változókat a struktúra definíciójában
ebben az esetben a címke opcionális, de hasznos, ha később akarunk hivatkozni a struktúrára

Struktúrák


struct kulcsszó
címke
példa
struct szemely
{
char nev[256];
int id;
float fizetes;
} sz1, sz2;
tagok ↔ a struktúra mezői
!
az sz1 és sz2 változók típusa struct szemely
és NEM egyszerűen szemely
DE
definálhatunk egy szemely típust
példa
typedef struct
{
char nev[256];
int id;
float fizetes;
} szemely;
szemely sz1, sz2;
VAGY → ha már a struktúra korábban címkével volt definiálva
példa
typedef struct szemely szemely;

Struktúrák

példa

```
typedef struct
{
    char nev[256];
    int id;
    float fizetes;
} szemely;

szemely sz1, sz2;
```

- elemek elérése
  - tag-hozzáférési operátor .
  - struktúra pointer operátor -> (erről később)

```
strncpy(sz1.nev, "Pistike", 7);
sz1.id = 1001;
sz1.fizetes = 3456.78;
```

- értékadás — érvényes → **KIVÉVE** tömbök esetén

```
sz2 = sz1;
```

- összehasonlítás — érvénytelen → **HELYESEN** elemenként történik

```
sz1 == sz2;
sz1 != sz2;
```

- inicializálás — érvényes

```
sz1 = {"Pistike", 1001, 3456.78};
sz2 = {.id = 1002, .fizetes = 4567.89};
```

C99

- struktúra tömbök — egy vagy több dimenziós

```
szemely szemelyek[50];
```

Példa

alkalmazottak.c

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

typedef struct szemely
{
    char *nev;
    int id;
    float fizetes;
} szemelyek;

void adatokFeltoltese(struct szemely alkalmazottak[], int alkalmazottakSzama)
{
    int i;
    int egesz, tizedes;
    for(i = 0; i < alkalmazottakSzama; i++)
    {
        egesz = rand()%3654 + 1346;
        tizedes = rand()%100;
        alkalmazottak[i].fizetes = egesz + (float)tizedes/100;
    }
}

float atlag(szemelyek alkalmazottak[], int alkalmazottakSzama)
{
    int i;
    float osszeg;
    for(i = 0; i < alkalmazottakSzama; i++)
    {
        osszeg += alkalmazottak[i].fizetes;
    }

    return osszeg/alkalmazottakSzama;
}

int main()
{
    szemelyek tmp, alkalmazottak[50];
    int alkalmazottakSzama = 50;

    tmp.nev = "Pistike";
    tmp.id = 1001;
    tmp.fizetes = 3456.78;

    adatokFeltoltese(alkalmazottak, alkalmazottakSzama);

    float atlagfizetes = atlag(alkalmazottak, alkalmazottakSzama);

    printf("> az atlagfizetes: %f\n", atlagfizetes);

    return 0;
}
```

struktúra deklarálása és az új típus definíciója

kivehető egy külön header állományba

FIGYELEM nem történt memóriafoglalás

később dinamikusan kell foglalni

deklarálás a struktúra címkét felhasználva

deklarálás az új típust felhasználva

struktúra tömb

???

kvíz 2. kérdés

1251 9558

Struktúrák

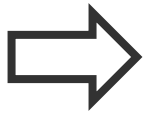
- amennyiben a kompilálás pillanatában nem ismerjük a szükséges struktúra tömb méretét, deklarálhatjuk ezt is dinamikusan

```
struct szemely *alkalmazottak;
alkalmazottak = (struct szemely *)calloc(alkalmazottakSzama, sizeof(struct szemely));
```

- a **nev** tagja a struktúrának egy pointer, de nincs memória foglalva neki
  - alternatív megoldás, hogy egy előre definiált maximális méretre lefoglaljuk azt

```
#define SMAX 256

typedef struct szemely
{
    char nev[SMAX];
    int id;
    float fizetes;
} szemelyek;
```



```
tmp.nev = "Pistike";
```

```
strncpy(tmp.nev, "Pistike", 7);
```

Struktúrára mutató pointer

példa `struct szemely *p = &tmp;`

→ az elemeit a pointer dereferenciájával érjük el

`(*p).fizetes` → fontos a zárójel használata, mert a `.` operátornak nagyobb a prioritása, mint a `*` operátornak

↓ ekvivalens művelet

`p->fizetes` → **ELŐNYE** 2 karakterrel rövidebb

- **ELŐNYE** a struktúrák sokszor nagy mennyiségű adatot tárolnak, ezért ha függvényben szeretnénk feldolgozni őket, előnyösebb pointerként átadni, mint érték szerint, mert ebben az esetben elkerüljük az elemenként történő másolást

Példa

komplex.c

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


typedef struct komplex
{
    double valos;
    double kepzetes;
} komplex;


void osszeg(komplex *osszeg, komplex *a, komplex *b)
{
    osszeg->valos = a->valos + b->valos;
    osszeg->kepzetes = a->kepzetes + b->kepzetes;
}


int main()
{
    komplex c1, c2, c3;

    c2.valos = 12.3;
    c2.kepzetes = 4.56;

    c3.valos = 45.6;
    c3.kepzetes = 7.89;

    osszeg(&c1, &c2, &c3);

    printf("> valos reszek osszege: %f\n", c1.valos);
    printf("> kepzetes reszek osszege: %f\n", c1.kepzetes);

    return 0;
}
```

Struktúra, mint stuktúra tagja

- a struktúrák elemeiként megjelenhet más struktúra is

példa

```
struct reszleg
{
    char nev[256];
    int id;
};


typedef struct alkalmazott
{
    char nev[256];
    int id;
    struct reszleg reszleg;
    struct cim *pCim;
    double fizetes;
} alkalmazott;
```

használat előtt definiálva kell legyenek, mert a fordítóprogramnak ismernie kell a struktúra méretét

a struktúrára mutató pointer mérete ismert, hisz az egy cím, ezért a `cim` struktúrát definiálhatjuk akár utólag is

Struktúrák és függvények

- a struktúrákat átadhatjuk függvényeknek, amelyek feldolgozhatják, módosíthatják a struktúra által tárolt adatokat
- az átadás 2 féle képpen történhet
  - érték szerinti átadás
  - csak egy pointert adunk át a függvénynek, amely a struktúrára mutat
- érték szerinti átadás esetén a függvény visszatérési értéke a módosított struktúra
- amennyiben egy pointert adunk át a függvénynek, az nem térít vissza semmit, hisz a pointeren keresztül közvetlenül módosítja a struktúrát

Struktúrák és függvények

érték szerint

példa

```
alkalmazott modosit1(alkalmazott e)
{
    int id;
    printf("> add meg a reszleg azonositojat : ");
    scanf("%d", &id);
    e.reszleg.id = id;

    return e;
}

e = modosit1(e);
```

- érték szerint átadjuk az **e** struktúrát a függvénynek, amely módosítja és visszatéríti azt
- átmásolja az **e** struktúra tartalmát az **e** formális paraméterbe
- a műveletek végrehajtása után az **e** formális paraméter által tárolt struktúrát visszamásolja az eredeti **e** struktúrába

pointeren keresztül

példa

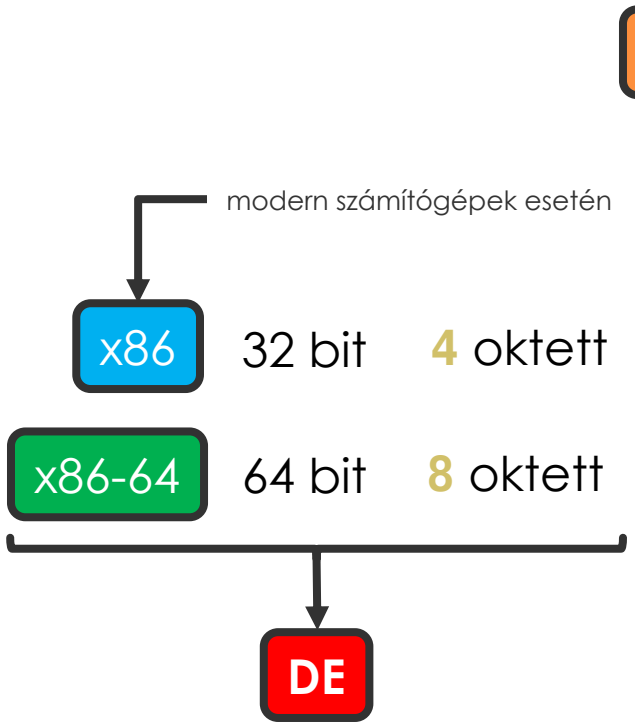
```
void modosit2(alkalmazott *e)
{
    int id;
    printf("> add meg a reszleg azonositojat : ");
    scanf("%d", &id);
    e->reszleg.id = id;
}


modosit2(&e);
```

- ez az **optimálisabb** megoldás
- a függvénynek csak egy pointert adunk át, amely az **e** struktúrára mutat
- a műveletek végrehajtása során a függvény közvetlenül az **e** struktúrát módosítja, nem történik semmilyen másolás

Memóriacímzés
általánosan

- a memóriacímzés **memóriaszó**nak nevezett egységekben történik



word

- az az alapegység, amelyet a processzor használ
- egy rögzített méretű adatmennyiség, amelyet az **utasításkészlet** vagy a **processzor hardware** egy egységként kezel
- a bitek száma, amely megszabja a méretét/szélességét/hosszát egy jellemző tulajdonsága bármely processzor designnak vagy számítógép architektúrának
- mérete a számítógép struktúrájának és működésének számos vonatkozásban tükröződik
- ekkora a processzor regisztereinek mérete: a memóriának egy műveletben átadható vagy onnan átvehető adatmennyiség

szinte az összes processzor architektúra esetén

Memóriacímzés

általánosan

DE — modern számítógépek

lehetővé teszik az adatok különböző egységekben történő kezelését

- 1 bit → legalább az alapegység méretéig
- esetenként akár ennél nagyobb egységekben is 16 → 64 oktett
  - speciális utasításkészlet segítségével

Memóriacímzés
az adatstruktúra igazítása

- az adatok rendezése és elérése a számítógép memóriájában
- 3 különálló, de szorosan összefüggő műveletből áll

adatok igazítása — data alignment

adatstruktúra kitöltése — data structure padding

csomagolás — packing

- a modern számítógépek az olvasást és írást akkor hajtják végre a leghatékonyabban, hogyha az adatok **természetesen igazodnak** a memóriában
  - az adatok memóriacíme az alapegység méretének többszöröse

1251 9558

kvíz 3. kérdés

példa

`short`	2 oktett	többszörösére illeszkedik
`int`	4 oktett	
`double`	8 oktett	

x86

Memóriacímzés

az adatstruktúra igazítása

adatok igazítása

data alignment

- feltételezi az elemek természetes módon való illeszkedését a memóriában
- azaz, az adatokat úgy helyezi el a memóriában, hogy annak kezdőcíme a természetes méret többszöröse legyen
- a lehető legkevesebb olvasási ciklussal teszi lehetővé az adatok memóriából való betöltését
- a természetes illeszkedés megvalósításához szükség lehet az **adatstruktúra** megfelelő **kitöltés**ére

adatstruktúra kitöltése

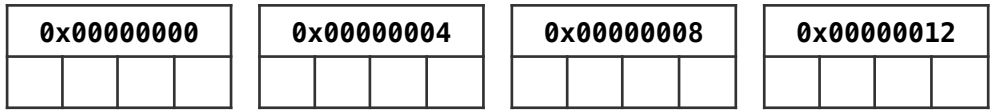
data structure padding

a megfelelő illesztés érdekében esetenként néhány **betét** beszúrása szükséges a szerkezet elemei közé vagy annak utolsó eleme után

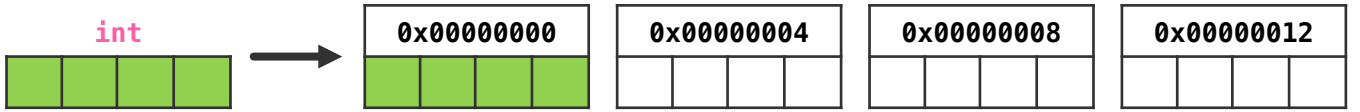
padding

kitöltő oktett

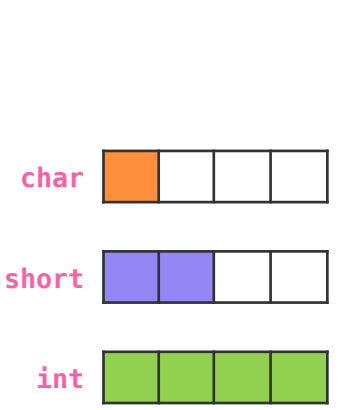
Memóriacímzés
az adatstruktúra igazítása

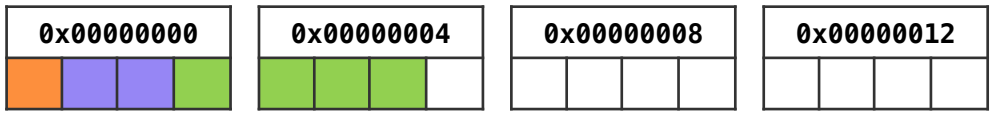


4 memóriaszó hosszúságú cella



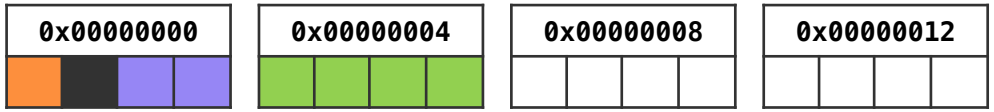
ugyan az a memóriaszegmens egy `int` típusú adattal





helytelen illesztés → 2 olvasási ciklust és biteltolást igényel, hogy az `int` típusú adatot is be tudjuk olvasni

helyes illesztés kitöltő betétet használva →
- természetes illesztés
- megnövekedett olvasási hatékonyság
- nagyobb memóriahasználat

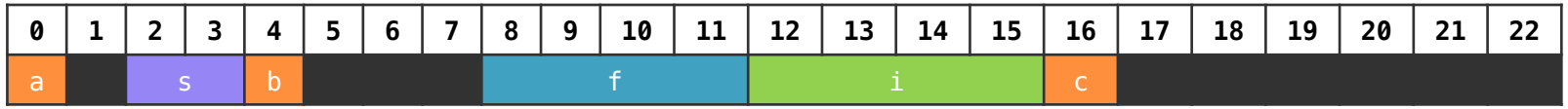


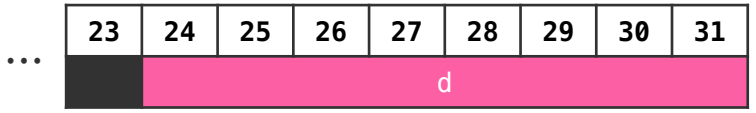
Memóriacímzés
az adatstruktúra igazítása

- a megfelelő adatstruktúra igazítás esetenként többlet memóriahasználatot eredményez
- ennek elkerülésére javasolt a struktúra tagjainak az átrendezése
  - első helyre helyezzük a legtöbb oktettet foglaló komponenst, majd azt követően méret szerinti csökkenő sorrendben a többit

példa

```
struct s1
{
    char a;
    short s;
    char b;
    float f;
    int i;
    char c;
    double d;
};
```





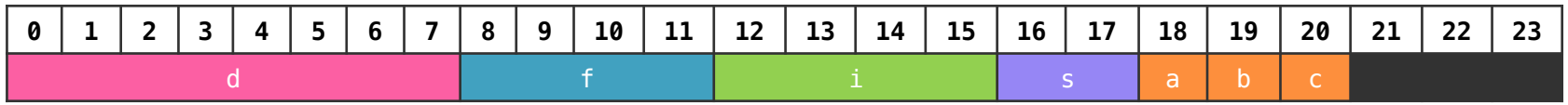
→ 11 kitöltő oktettet tartalmaz

→ összesen 32 oktettet foglal

! a kitöltő betét csak akkor kerül beillesztésre, ha struktúra tagját egy nagyobb igazítási igényű tag követi vagy a szerkezet végén

példa

```
struct s2
{
    double d;
    float f;
    int i;
    short s;
    char a, b, c;
};
```



→ 3 kitöltő oktettet tartalmaz

→ összesen 24 oktettet foglal

! kis átrendezés, óriási ugrás hatékonyságban

Struktúrák

meghatározott hosszúságú tagok

- a **C** nyelv lehetőséget ad a programozónak megszabni, hogy egy struktúra tagja hány biten kerüljön ábrázolásra, eltárolásra
  - kizárólag csak az `int` és `unsigned int` típusok esetén használható
  - hatékonyabb memóriahasználatot eredményez
    - főként, minkor ismert a változó értékének értelmezési tartománya
    - általában kis értékek esetén

példa

```
struct datum
{
    unsigned int ev;
    unsigned int honap:4;
    unsigned int nap:5;
};
```

az év hónapjai maximálisan 4 biten ábrázolhatók

1–12 között vehet fel értékeket

a hónap napjai maximálisan 5 biten ábrázolhatók

1–31 között vehet fel értékeket

Példa

datum.c

nincsen meghatározva a struktúra tagjainak bit-hosszúsága

```c
#include <stdio.h>

struct datum
{
    unsigned int ev;
    unsigned int honap;
    unsigned int nap;
};

int main()
{
    printf("> struktura merete oktettekben: %lu\n", sizeof(struct datum));
    struct datum ma = {2024, 1, 8};
    printf("> a mai datum: %d.%d.%d.\n", ma.ev, ma.honap, ma.nap);

    return 0;
}
```

```
> a struktura merete oktettekben: 12
> a mai datum: 2024.1.8.
```

Példa

datumFixed.c

meghatározott a tagok hossza

```c
#include <stdio.h>

struct datum
{
    unsigned int ev;
    unsigned int honap:4;
    unsigned int nap:5;
};

int main()
{
    printf("> struktura merete oktettekben: %lu\n", sizeof(struct datum));
    struct datum ma = {2024, 1, 8};
    printf("> a mai datum: %d.%d.%d.\n", ma.ev, ma.honap, ma.nap);

    return 0;
}
```

```
> a struktura merete oktettekben: 8
> a mai datum: 2024.1.8.
```

Struktúrák

meghatározott hosszúságú tagok

- ezekben az esetekben a **C** nyelv nem egyértelműsíti, hogy milyen sorrendben kerülnek kiosztásra a memóriacímek
- a kiosztás függhet a rendszer architektúrájától és/vagy a fordítóprogramtól is
- ezért mielőtt bármit is feltételeznénk, érdemes egyszerű példákon keresztül ellenőrizni

adatok igazítása → ha egy olyan tag, amely meghatározott bit hosszúsággal van definiálva, kisebb dimenzióval rendelkezik, mint egy `int`, akkor **kitöltés**re kerülnek a fennmaradó bitek → **adatstruktúra kitöltése**

data alignment

- használhatunk **név nélküli tagok**at a struktúrákban
  - explicit adatstruktúra kitöltést végeznek → **data structure padding**
  - nem tárolhatnak információt
  - `0` bit hosszúságú név nélküli tagot követő tag a következő olyan címen fog kezdődni, amelyen az a típus eltárolható

Struktúrák

meghatározott hosszúságú tagok

példa

```
struct on_off
{
    unsigned light:1;
    unsigned toaster:1;
    int count;
    unsigned ac:4;
    unsigned:4;
    unsigned clock:1;
    unsigned:0;
    unsigned flag:1;
} kitchen;
```

feltételezzük, hogy az `int` az `4` oktettet foglal

a **kitchen** struktúra így **16** oktetten ábrázolható

tag név	elfoglalt tárhely bit	
light	1	
toaster	1	
padding	30	→ a legközelebbi `int` határáig
count	32	
ac	4	
név nélküli tag	4	→ `4` bittel való eltolás/párnázás
clock	1	
név nélküli tag	23	→ a legközelebbi `int` határáig
flag	1	
padding	31	→ a legközelebbi `int` határáig

Struktúrák
meghatározott hosszúságú tagok

- **KORLÁTOZÁSOK**
  - maximálisan 64 bit hosszúságú mezőket definiálhatunk
  - kompatibilitás és hordozhatóság kedvéért azonban ajánlott maximálisan 32 bit hosszúságú mezőkkel dolgozni
  - nem használhatunk előre meghatározott hosszúságú tagokat tömbök esetén
  - nem hivatkozhatunk az ilyen formában lerögzített név nélküli mezők címére
  - nem definiálhatunk pointert, ami egy név nélküli mezőre mutat
  - amennyiben egy nagyobb értékkel próbálunk egy ilyen tagot inicializálni, a legkisebb jelentőségű biteket tartja meg, amelyek a megszabott bit hosszúságon ábrázolhatók

Memóriacímzés
az adatstruktúra igazítása

- a **C** nyelv nem engedi meg, hogy a fordító helyet takarítson meg a tagok megfelelő sorrendezésével, más nyelvek viszont igen
- **DE** a legtöbb **C** fordítónak meg lehet mondani, hogy **csomagolják** a struktúra tagjait egy bizonyos igazítási szintre
  - *pl.* `pack(2)` → az 1 oktettnél nagyobb tagok egy 2 oktettes határhoz igazodnak → a kitöltő betét maximálisan 1 oktett hosszúságú lehet

csomagolás

→ elsődleges célja a memóriamegőrzés

packing

→ *pl.* egy 1 oktett méretű és egy 4 oktett méretű tagot tartalmazó struktúra megfelelő igazításához 3 kitöltő oktettre van szükség

→ egy ilyen struktúra nagy tömbje 37.5%-val kevesebb memóriát használna, ha csomagolva lenne

→ **VISZONT** az elemek elérése hosszabb időt venne igénybe

tér-idő kompromisszum

→ felhasználható adatstruktúra formázásra, továbbítás céljából szabványos protokollok használatával

példa

```
#pragma pack(2)
struct struktura
{
    char tag;
    int elem;
};
```

A felsorolás típus

enum

- akkor használjuk, amikor valamely változónak több mint 2, de véges (nem túl sok) értéket kívánunk tárolni
- az **enum** tagjai egész típusú változók/értékek, amelyekhez szimbolikus neveket kapcsolunk
- első kiadásakor a **C** nyelv nem rendelkezett felsorolás típussal, ez később került bele
- **C** nyelvben a felsorlásokat explicit definiálásokkal hozzuk létre, viszont **NEM** eredményez memóriafoglalást
- szintaxisa hasonló a **struct** szintaxisához

példa

```
enum napok
{
    hetfo,
    kedd,
    szerda,
    csutortok,
    pentek,
    szombat,
    vasarnap
};
```

- nem használunk ékezeteket, sem idézőjeleket, mert ezek változó nevek
- ezen változónevekhez implicit egész értékeket rendel a fordítóprogram `const int`
- az implicit értékek ebben az esetben {0, 1, 2, 3, 4, 5, 6}
- viszont meghatározhatjuk egy vagy több tagnak is az értékét → `hetfo = 1,`
- a változtatás kihat az őt követő összes változó értékére → {1, 2, 3, 4, 5, 6, 7}

A felsorolás típus

enum

- hasznos, hisz csökkenti a használandó `#define` direktívák számát
- az **`enum`** által tárolt értékek abban a blokkban vagy függvényben érhetők el, amelyikben definiálva voltak
- a deklaráció és használat hasonló a **`struct`** szerkezetek használatához

példa

```
enum napok
{
    hetfo,
    kedd,
    szerda,
    csutortok,
    pentek,
    szombat,
    vasarnap
};

enum napok nap1, nap2;

...

    if(nap1 != nap2)
    {
        ...
    }
```

a típus **`enum`** napok

`typedef` segítségével definiálhatunk egy új típust

egyszerüsíti a használatot

példa

```
typedef enum napok
{
    hetfo,
    kedd,
    szerda,
    csutortok,
    pentek,
    szombat,
    vasarnap
} napok;

napok nap1, nap2;

...

    if(nap1 != nap2)
    {
        ...
    }
```

1251 9558 kvíz 4. kérdés

Az unió típus

`union`

- formailag megegyezik a struktúrával
- **DE**
  - a struktúra elemei egymást követő, különböző memóriaszegmenseken vannak eltárolva
  - az unió típus tagjai **azonos memóriaterületen** helyezkednek el

    → rendszerint struktúrák
- mérete a legnagyobb tag méretével fog megegyezni
- **FŐ CÉLJA** ugyan azt a memóriaterületet a program különböző időpontokban különböző célokra használja
- leggyakrabban rendszerprogramokban fordul elő, felhasználói programokban nagyon ritka

Példa

union.c

1251 9558

kvíz 5. kérdés

```c
#include <stdio.h>
#include <string.h>
#define N 2

typedef union mennyiseg
{
    int darab;
    double kg;
} mennyiseg;

typedef struct termek
{
    char nev[256];
    float egysegnyiAr;
    int mertekegyseg;
    mennyiseg m;
} termek;

int main()
{
    termek alma, labda;
    termek *polc[N];

    strcpy(alma.nev, "golden");
    alma.egysegnyiAr = 6.34;
    alma.mertekegyseg = 1;
    alma.m.kg = 56.78;

    strcpy(labda.nev, "kosarlabda");
    labda.egysegnyiAr = 54.99;
    labda.mertekegyseg = 2;
    labda.m.darab = 5;

    polc[0] = &alma;
    polc[1] = &labda;

    for(int i = 0; i < N; i++)
    {
        printf("> termekinformacio : %s\n", polc[i]->nev);
        switch(polc[i]->mertekegyseg)
        {
            case 1:
                printf(">> %f kg van raktaron\n", polc[i]->m.kg);
                break;
            case 2:
                printf(">> %d db van raktaron\n", polc[i]->m.darab);
        }
    }

    return 0;
}
```

További részletek
bibliográfia

- K. N. King C programming – A modern approach, 2nd edition, W. W. Norton & Co., 2008
  - 16. fejezet
- Deitel & Deitel C How to Program, 6th edition, Pearson, 2009
  - 10. fejezet