1. feladat: Milyen hibákat talál az alábbi megoldásokban?

```
m = malloc(80);
m = NULL;
```

megoldás: 80 bájt lefoglalása, majd a pointer nullázása. Probléma: - a lefoglalt 80 byte nem szabadul fel. `m = NULL` sor természetesen nullázza az `m` mutatót, de ettől még a 80 bájt lefoglalva marad.

Egy pointerhez memória lefoglalása majd amikor már nincs rá szükség, akkor a felszabadítása az alábbi váz alapján valósítható meg.

```
m = malloc(80);
// számolás.. egyéb kódok
free(m);
```

Nézzük a következő hibás példát. Mi lehet a hiba?

```
free(n);
n = 5;
```

megoldás: felszabadított memóriaterületre akarunk írni. Ha már meghívtuk a free() memória felszabadító függvényt, akkor nem lehet tovább használni a pointert. (ameddig a malloc()-al újra foglalunk neki memóriát.)

A következő példában nem használunk dinamikus memóriafoglalást, ami elvileg még jó is lehet, de mégsem szabályos. Miért?

```
char *p;
*p = 'a';
```

megoldás: nem inicializált pointert akarunk használni. Azaz a `char *p`, nem ad értéket a `p` pointernek, csak helyet foglal neki, és azon a helyen 0 van, vagy valami memóriaszemét. Ha a következő sorban a `*p = 'a'` -val a p által mutatott címre bele szeretnénk másolni a 'a'-t akkor az nem lesz lehetséges, mert a 0-ás címre nem írhatunk.

Feladat:: foglaljunk le dinamikusan egy 10 elemű int vektort.

```
#include <stdlib.h>

int main() {
    int *v = (int *)malloc(1000 * sizeof(int));
    free(v);
    return 0;
}
```

A kód igazából nem csinál túl sok használhatót, csak demonstrálja a malloc() használatát. A paramétere azt jelenti, hogy hány bájtot szeretnénk lefoglalni. Önmagában 1000-el meghívva csak 250 darab int értéknek foglal helyet, mert minden `int` 4 byte-ot foglal. Mivel ezt nem akarjuk fejből tudni ezért használjuk a beépített sizeof() operátort, ami sizeof(int) esetén 4-et ad vissza. Összefoglalva az 1000 int lefoglalásához 4000 byte-ok kell lefoglalni.

Feladat:: Az alábbi 3 függvényből melyik helyes, illetve hibás?

```
int* f(void)
{
  int x = 10;
  return (&x);
}
int* g(void)
{
  int * p;
  *p = 10;
  return p;
}
int* h(void)
{
  int *p;
  p = (int *) malloc (sizeof(int));
  return p;
}
```

megoldás: csak a h() függvény helyes. A f() és g() hibásak, mert ezekben az esetekben a memóriacím a veremben jön létre és a visszaadott érték helytelen memóriacímre fog mutatni. Az f() és g() esetén lokális változókat hozunk létre, és ezekre mutató memóriacímet adunk vissza. A függvény használatakor, a használat után ezek a memóriacímek nem használhatók tovább. A h() esetén is a p változó lokális lesz, az is megszűnik a függvény használata után, de a malloc() által

visszaadott memória cím (mint számérték továbbra is használható) a `return p` itt is nem a p-t, hanem a p-ben tárolt címet adja vissza, ami a h() esetén továbbra is használható.

A példa azt mutatja, hogy a malloc() által létrehozott cím és a mögötte lefoglalt memória globális, nem számít hol hívtuk meg.

Feladat:: Mi lesz az alábbi program kimenete?

```
#include<stdio.h>

void f(int *a)
{
    a = (int*)malloc(sizeof(int));
}

int main()
{
    int *p;
    f(p);
    *p = 10;
    printf("%d",*p);
}
```

megoldás: nem ír ki semmit, hanem lefagy.

Feladat:: Hogyan lehetne kijavítani az előző feladatban szereplő programot?

megoldás: az előző program azért nem működik, mert a pointer nincs inicializálva (csak egy másolat), ezért a 0-s memóriacímet adja át az f() függvénynek, így a malloc lefoglalja a 4 byte-ot, de hibás helyre írja vissza. A javítás során vegyük figyelembe, hogy egy pointerre mutató pointer már ténylegesen a p változó címét adja vissza, amibe már a f() malloc()-ja már be tudja írni a lefoglalt memória címét.

```
#include<stdio.h>

void f(int **a)
{
    *a = (int*)malloc(sizeof(int));
}

int main()
{
    int *p;
```

```
    f(&p);
    *p = 10;
    printf("%d",*p);
}
```

Feladat:: Mi a probléma következő programmal?

```
#include<stdio.h>

int main()
{
    float *p = (float *)malloc(sizeof(float));
    p = NULL;

    free(p);
}
```

megoldás: a p NULL-ázása után, a free() nem tudja, hogy hol van az a memóriacím, amit fel kell szabadítani, a free() igazából nem egy változót magát, hanem a benne tárolt memória címet (mint számértékhez tartozó foglalást) szünteti meg.) Más szóval: a free() nem a p változót, hanem az abban tárolt címet szabadítja fel.

Ellenőrző kérdések

Mi a helyes deklarációja egy int típusra mutató pointernek? A) int p*; B) int *p; C) pointer int p; D) int &p;

Megoldás: B

Hogyan lehet helyesen lefoglalni memóriát egy 10 elemű int tömbnek dinamikusan?

A) int *arr = malloc(10); B) int arr = malloc(10 * sizeof(int)); C) int *arr = malloc(10 * sizeof(int)); D) int arr[10] = malloc(sizeof(int));

Helyes válasz: C

Mi lesz az eredmény?

```
int array[] = {100, 200, 300};
int *ptr = array;
```

```
printf("%d", *(ptr++));
printf("%d", *ptr);
```

A) 100200 B) 200300 C) 100100 D) 200200

Megoldás: A. *(ptr++) előbb kiírja a 100-at, majd eggyel tovább lépteti a pointert. A következő kiírás már a 200-ra mutat.

Melyik állítás igaz az alábbiak közül?

A) Egy pointer típusa nem függ attól, hogy milyen adatra mutat. B) Egy pointer mindig egész számot tárol. C) Pointerek aritmetikája nem megengedett C-ben. D) Egy pointer típusa meghatározza, hogy milyen típusú adatot érhetünk el rajta keresztül.

Helyes válasz: D

Mi lesz az eredmény?

```
int x = 5, y = 10;
int *p = &x;
int *q = &y;
*p = *q;
printf("%d %d", x, y);
```

A) 5 10 B) 10 10 C) 5 5 D) 10 5

Helyes válasz: B.

Mire mutat a p pointer a következő deklaráció után?

```
int arr[10];
int *p = arr;
```

A) Az arr tömb első elemére B) Az arr tömb utolsó elemére C) Az arr tömb méretére D) Véletlenszerű memória címre

Megoldás: A

Mi a hiba a következő kódrészletben?

```
int num = 20;
int *ptr;
*ptr = num;
```

A) Az arr tömb első elemére B) Az arr tömb utolsó elemére C) Az arr tömb méretére D) Véletlenszerű memória címre

A) Érvénytelen pointer-dereferencia (nem inicializált pointer) B) Szintaktikai hiba C) Hiányzó pontosvessző D) A num változó helytelen típusa

Megoldás: A

Mi lesz a következő kód kimenete?

```
char *str = "Hello";
printf("%c", *(str+1));
```

A) H B) e C) l D) o

Megoldás: B

Mit jelent, ha egy pointer NULL értékű?

A) A pointer érvényes memória címre mutat B) A pointer egy egész számot tárol C) A pointer nem mutat érvényes memória címre D) A pointer konstans értékre mutat

Megoldás: C

Mi lesz a következő kód kimenete?

```
int a[] = {2, 4, 6, 8};
int *p = a;
printf("%d", *(p + 2));
```

A) 2 B) 4 C) 6 D) 8

Megoldás: C

Melyik operátor adja vissza egy változó memória-címét?

A) * B) & C) % D) #

Megoldás: B

Mit ír ki a következő programrészlet?

```
int x = 5;
int *p = &x;
*p = 10;
printf("%d", x);
```

A) 5 B) 0 C) 10 D) véletlenszerű érték

Megoldás: C , mivel az x értéke megváltozik a pointeren keresztül