```
PROGRAMOZÁS 2 LABOR plusz PÉLDÁK, 'AVR C' és 'AVR ASM' finomságok
====================================================================
1. RÉSZ:   -1111-1111-1111-1111-1111-1111-1111-1111-1111-1111-1111

/*
====================================================================
1. PÉLDA:

      - AVR-C fejlesztőkörnyezet alatt,
        ASM utasítások használatának bemutatása C-kódban

      - DEBUG-olás (sárga nyíl léptetése) működik, azaz
        az egyes lépések követése szimulátorban lehetséges (!!!)

      - C-kódot írunk és abba ágyazzuk bele az ASM utasításokat,
        ahol arra szükség van.

        Az ASM-kód beágyazás (soronkénti ASM-kód) akkor működik, ha az
        egyes ASM utasítások egy blokkban vannak, azaz az egyes logikai
        ASM lépések között nincs C utasítás.

      - MEGVALÓSÍTANDÓ FELADAT:
         ADAT (DATA) MEMÓRIA feltöltése ASM utasítások segítségével
         (ADAT memória ablak figyelése, ahogy sima ASM kód ellenőrzésekor)

         - 1. lépésben:
           ADAT (DATA) MEMÓRIA feltöltése 0x200 memóriacímtől felfelé
           0x301-től növekvő 10 darab számmal
           ( 0x101, 0x102, 0x103, 0x104, 0x105,
             0x106, 0x107, 0x108, 0x109, 0x10A )

         - 2. lépésben:
            ADAT (DATA) MEMÓRIA visszaolvasása és a visszaolvasott
            adatok összegzése

            AZ ÖSSZEGZETT EREDMÉNY ALSÓ BYTE-jának KIÍRÁSA A LED-ekre
             ( 0x101 + 0x102 + 0x103 + 0x104 + 0x105 +
               0x106 + 0x107 + 0x108 + 0x109 + 0x10A = 0xA37
               =>  alsó BYTE: 0x37  =>  0b_0011_0111 [eredmény] )

====================================================================
*/

#include <avr/io.h>

void mem_iras(void);
void mem_olvasas(void);

int main()
{           // C-kódot írunk és abba ágyazzuk bele az ASM utasításokat,
            // ahol arra szükség van

      asm (" nop ");
      DDRB=DDRD=0xF0;         // LED-ek lábai KI-menetre állítása,

      mem_iras();             // MEMÓRIA ÍRÁSA külön függvényben
      mem_olvasas();          // MEMÓRIA OLVASÁSA külön függvényben

      asm(" nop ");           // BAB

while(1){};                   // BAB
return 0;
};
```

```
// ---- függvények ---- ---- ---- ---- ---- ----

void mem_iras(void)
{
      asm (" nop ");

      asm (" ldi r16, 10 ");              // számláló
      asm (" ldi r19,  0 ");              // segédregiszter HIGH
      asm (" ldi r18,  1 ");              // segédregiszter LOW

      asm (" ldi r31, 0x02 ");            // MEMÓRIA KEZDŐCÍME (HIGH)
      asm (" ldi r30, 0x00 ");            // (LOW)

      asm (" ldi r25, 0x01 ");            // ADAT KEZDŐÉRTÉKE  (HIGH)
      asm (" ldi r24, 0x01 ");            // (LOW)

      asm (" ciklus_ir: ");

            asm (" st Z+, r24 " );        // MEMÓRIA írása
            asm (" st Z+, r25 " );

            asm (" add r24, r18 " );      // számérték növelése (macerásan)
            asm (" adc r25, r19 " );

            asm (" dec r16 ");            // számláló csökkentése

      asm (" brne ciklus_ir ");

      asm (" nop ");
}


void mem_olvasas(void)
{
      asm (" nop ");

      asm (" ldi r16, 10 ");  // számláló
      asm (" clr r19 ");      // segédregiszter HIGH (most az összegzéshez)
      asm (" clr r18 ");      // segédregiszter LOW  (most az összegzéshez)

      asm (" ldi r31, 0x02 ");            // MEMÓRIA KEZDŐCÍME (HIGH)
      asm (" ldi r30, 0x00 ");            // (LOW)

      asm (" ciklus_olvas: ");

            asm (" ld r24, Z+ " );        // MEMÓRIA olvasása
            asm (" ld r25, Z+ " );

            asm (" add r18, r24 " );      // összegzés
            asm (" adc r19, r25 " );      // (eredmény az r19:r18-ben)

            asm (" dec r16 ");            // számláló csökkentése

      asm (" brne ciklus_olvas ");

            // az összegzett eredmény ALSÓ BYTE-jának kiíratása a LED-ekre
            // =>  alsó BYTE: 0x37  =>  0b_0011_0111 [eredmény]

      asm (" out 0x12, r18 ");            // out PORTD(címe: 0x12), r18
      asm (" swap r18 ");
      asm (" out 0x18, r18 ");            // out PORTB(címe: 0x18), r18
      asm (" swap r18 ");

      asm (" nop ");
}
```

```
/*
==========================================================================
2. PÉLDA:

      - AVR-C fejlesztőkörnyezet alatt,
        ADAT (DATA) MEMÓRIA használatának bemutatása, C-kód írásakor
        (ADAT memória ablak figyelése)

      - DEBUG-olás (sárga nyíl léptetése) működik, azaz
        az egyes lépések követése szimulátorban lehetséges (!!!)

      - nyomon követjük az egyes VÁLTOZÓK értékét az ADAT-MEMÓRIÁBAN, és
        nyomon követjük a TÖMBÖK-nek lefoglalt helyet az ADAT-MEMÓRIÁBAN

      - ADAT (DATA) MEMÓRIA feltöltése C-kód és TÖMBÖK segítségével
        UGYANÚGY MINTHA SIMA C-kódot írnánk

      !!!!
```

```
#include <avr/io.h>                             // 2. PÉLDA (előző oldal)


int main()
{                                   // deklarációk, volatile FONTOS
      volatile int i=0, szam=0, osszeg=0, vege=0;

      volatile int tomb_1[20]={0};
      volatile int tomb_2[20]={0};

      DDRB=DDRD=0xF0;               // LED-ek lábai KI-menetre állítása,


            // ------------------------------------------------------------
      i=0;
      szam=0x101;       // kezdőszám

      while(i<10)       // 'tomb_1' feltöltése a adatokkal
      {
            tomb_1[i]=szam;
            szam++;
            i++;
      }


            // ------------------------------------------------------------
      i=0;

      while(i<10)       // 'tomb_1'-ből az adatok áttöltése 'tomb_2'-be
      {
            tomb_2[i]=tomb_1[i];
            i++;
      }


            // ------------------------------------------------------------
      i=0;
      osszeg=0;

      while(i<10)       // 'tomb_2'-ből az adatok kiolvasása és összegzése
      {
            szam=tomb_1[i];
            osszeg=osszeg+szam;
            i++;
      }


            // -- AZ ÖSSZEGZETT EREDMÉNY ALSÓ BYTE-jának KIÍRÁSA A LED-ekre
            //    =>  alsó BYTE: 0x37  =>  0b_0011_0111 [eredmény]

      PORTD  =  osszeg;
      PORTB  =( osszeg << 4);


      vege++; // Kell, hogy lássuk a PORT-ok bebillenését a szimulátorban
      vege++; // Kell, hogy lássuk a PORT-ok bebillenését a szimulátorban


while(1){}; // BAB, hogy FIX vége legyen a programunknak
return 0;
};
```

```
/*
==========================================================================
3. PÉLDA:

      - AVR-C fejlesztőkörnyezet alatt,
        ADAT (DATA) MEMÓRIA használatának bemutatása, C-kód írásakor
        (ADAT memória ablak figyelése)

      - DEBUG-olás (sárga nyíl léptetése) működik, azaz
        az egyes lépések követése szimulátorban lehetséges (!!!)

      - nyomon követjük az egyes VÁLTOZÓK értékét az ADAT-MEMÓRIÁBAN, és
        nyomon követjük a TÖMBÖK-nek lefoglalt helyet az ADAT-MEMÓRIÁBAN

      - ADAT (DATA) MEMÓRIA feltöltése C-kód és TÖMBÖK segítségével
        UGYANÚGY MINTHA SIMA C-kódot írnánk,

      !!!!
```

```
#include <avr/io.h>                             // 3. PÉLDA (előző oldal)

void fgv1_feltolt (volatile int *p_t1_f);
void fgv2_masol   (volatile int *p_t1_f, volatile int *p_t2_f);
volatile int fgv3_olvas (volatile int *p_t2_f);

int main()
{                                   // deklarációk, volatile FONTOS
      volatile int semmi=0;         // csak azért kell, hogy lássuk a fgv
                                    // hívásokat és a végén a lezárást

      volatile int osszeg=0;


      volatile int tomb1[20]={0};
      volatile int tomb2[20]={0};

      volatile int *p_t1;
      volatile int *p_t2;


      DDRB=DDRD=0xF0;               // LED-ek lábai KI-menetre állítása


      semmi=1;                      // ..., hogy lássuk


      p_t1 = &tomb1[0];
      fgv1_feltolt( p_t1 );         // fgv hívás


      semmi=2;                      // ..., hogy lássuk


      p_t1 = &tomb1[0];
      p_t2 = &tomb2[0];
      fgv2_masol( p_t1, p_t2 );     // fgv hívás


      semmi=3;                      // ..., hogy lássuk


      p_t2 = &tomb2[0];
      osszeg = fgv3_olvas( p_t2 );  // fgv hívás


      semmi=4;                      // ..., hogy lássuk


            // AZ ÖSSZEGZETT EREDMÉNY ALSÓ BYTE-jának KIÍRÁSA A LED-ekre
            // =>  alsó BYTE: 0x37  =>  0b_0011_0111 [eredmény]
      PORTD  =  osszeg;
      PORTB  =( osszeg << 4);


      semmi=5; // Kell, hogy lássuk a PORT-ok bebillenését a szimulátorban
      semmi=6; // Kell, hogy lássuk a PORT-ok bebillenését a szimulátorban


while(1){};       // BAB, hogy FIX vége legyen a programunknak
return 0;
};
```

```
// ---- függvények ---- ---- ---- ---- ---- ----

void fgv1_feltolt (volatile int *p_t1_f)
{
      volatile int i=0;
      volatile int szam=0x101;            // kezdőszám

      while(i<10)       // 'tomb_1' feltöltése az adatokkal
      {
            *p_t1_f=szam;

            p_t1_f++;
            szam++;
            i++;
      }
}



void fgv2_masol   (volatile int *p_t1_f, volatile int *p_t2_f)
{
      volatile int i=0;

      while(i<10)       // 'tomb_1'-ből az adatok áttöltése 'tomb_2'-be
      {
            *p_t2_f=*p_t1_f;

            p_t1_f++;
            p_t2_f++;
            i++;
      }
}



volatile int fgv3_olvas   (volatile int *p_t2_f)
{
      volatile int osszeg_f=0;
      volatile int i=0;

      while(i<10)       // 'tomb_2'-ből az adatok kiolvasása és összegzése
      {
            osszeg_f = osszeg_f + (*p_t2_f);
            p_t2_f++;
            i++;
      }

return osszeg_f;
}
```

==========
2. RÉSZ: -2222-2222-2222-2222-2222-2222-2222-2222-2222-2222-2222
További egyszerű PÉLDA, ugyanaz a feladat megoldva három féleképpen
---

FONTOS; - a uC C esetben, lehet nem debug-olni, és lehet debug-olni.

A ESET; NEM debug-olható a kód:

```
#include <avr/io.h>

int szamol(int a_x, int b_x);

int main()
{

    int a=4, b=2, c=0, semmi=0;
    c=szamol(a,b);

    DDRB=DDRD=0xF0;
    PORTD=c;
    PORTB=(c<<4);

    semmi++; // csak azért, hogy lássuk a
    semmi++; // PORT-beállításokat is
    semmi++;
    semmi++;

while(1){};
return 0;
}


int szamol(int a_x, int b_x)
{
    int c_vissza=0;

    c_vissza=a_x+b_x;

return c_vissza;
}
```

B ESET; debug-olható a kód:

- ha az összes szabad változó és az összes függvény változó elé is CSAK egy 'volatile' kiegészítést teszünk, akkor a RAM végére teszi a fejlesztőkörnyezet az értékeket DEBUG-oláskor.

  Debug-olni így már az CTRL+F7 és F11-el lehet.

```
#include <avr/io.h>

volatile int szamol(volatile int a_x, volatile int b_x);

int main()
{

    volatile int a=4, b=2, c=0, semmi=0;
    c=szamol(a,b);

    DDRB=DDRD=0xF0;
    PORTD=c;
    PORTB=(c<<4);

    semmi++; // csak azért, hogy lássuk a
    semmi++; // PORT-beállításokat is
    semmi++;
    semmi++;

while(1){};
return 0;
}


volatile int szamol(volatile int a_x, volatile int b_x)
{
    volatile int c_vissza=0;

    c_vissza=a_x+b_x;

return c_vissza;
}
```

C ESET; debug-olható kód, POINTER-rel megoldva:

- ha az összes szabad változó és az összes függvény változó és összes POINTER elé is CSAK egy 'volatile' kiegészítést teszünk, akkor a RAM végére teszi a fejlesztőkörnyezet az értékeket DEBUG-oláskor.

  Debug-olni így már az CTRL+F7 és F11-el lehet.

```
#include <avr/io.h>

volatile int szamol( volatile int *p1x, volatile int *p2x);

int main()
{
    volatile int a=4, b=2, c=0, semmi=0;
    volatile int *p1, *p2;    // POINTEREK
    p1=&a;
    p2=&b;
    c=szamol(p1,p2);

    DDRB=DDRD=0xF0;
    PORTD=c;
    PORTB=(c<<4);

    semmi++;   // csak azért, hogy lássuk a
    semmi++;   // PORT-beállításokat is
    semmi++;
    semmi++;

while(1){};
return 0;
}


volatile int szamol( volatile int *p1x, volatile int *p2x)
{
    volatile int c_vissza=0;

    c_vissza=*p1x+*p2x;

return c_vissza;
}
```

TEHÁT, ha minden mintapéldát meg akarjuk nézni debug-olva is, akkor a 'volatile' kiegészítést minden változó elé odaírjuk. Így már a fejlesztőkörnyezet nem fogja kioptimalizálni a változókat, és a változó értékek a RAM végére bekerülnek.