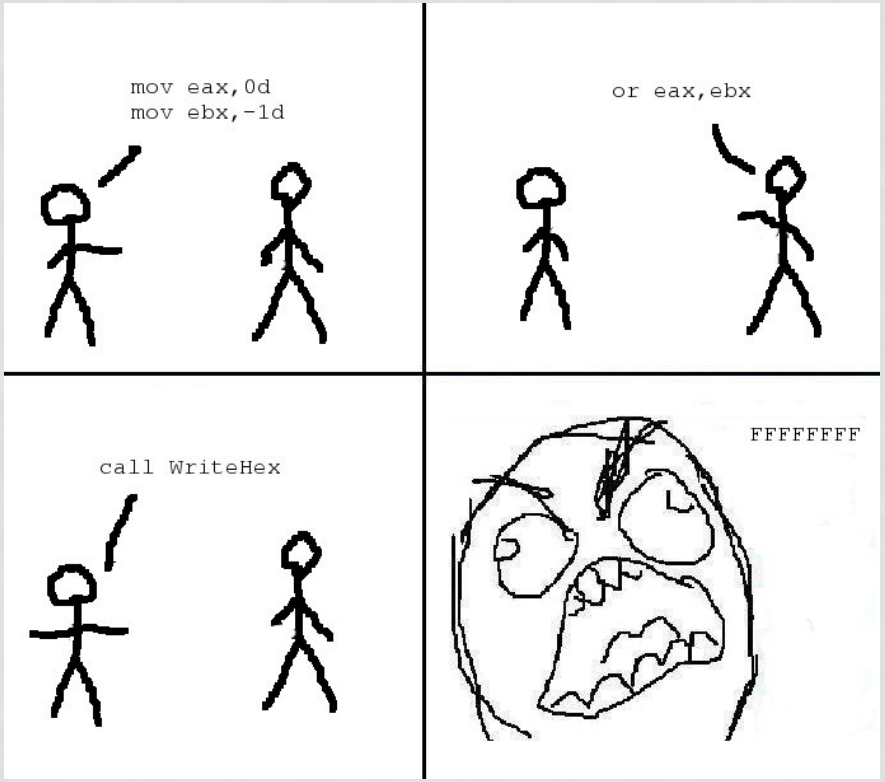


ASSEMBLY & C

WEEK 3-2

AGENDA

week	onderwerp	week	week
1	de structuur van AVR-assembly AVR instructies AVR registers en I/O ATmega memory map Atmel Studio AVR expressies en directives AVR addressing modes	3	de structuur van C-programma's ATMEL studio en AVR libc typen, constanten en operatoren AVR register access in C control statements functies & stackframe visibility scope arrays & strings struct & enum
2	flow of control spring instructies, control structuren Arduino UNO AVR studio stack & subroutines interrupts timer/counters switch bounce	4	interrupts in C TM1638 led&key UART PWM & ADC using a TTC-scheduler state diagram

AGENDA

- **control statements**
- functies & stackframe
- array & string
- structure
- enum
- local & global scope

CONTROL STATEMENTS

- zelfde structuren als in java
- `if/else`
- `return`
- `switch/case/default`
- `for (init; condition; loop-increment)`
- `while (condition) { statements }`
- `do { statements } while (condition)`
- `break & continue`

CONTROL STATEMENTS

```
 1  void update_leds()
 2  {
 3      uint8_t pwm_set = 0;
 4
 5      if (gv_switch_value != 0xff) {
 6          PORTB = gv_switch_value;
 7          for (int i=0; i < 8; i++) {
 8              if ((gv_switch_value & 1) == 0) {
 9                  pwm_set = i;
10                  break;
11              } else {
12                  gv_switch_value = gv_switch_value >> 1;
13              }
14          };
15      }
16      gv_update_flag = 0;
17  }
```

AGENDA

- control statements
- **functies & stackframe**
- array & string
- structure
- enum
- local & global scope

FUNCTIES

- modules in C :
  - files
  - functies
- nesten van functies mag niet
- main() is de eerste functie die wordt uitgevoerd

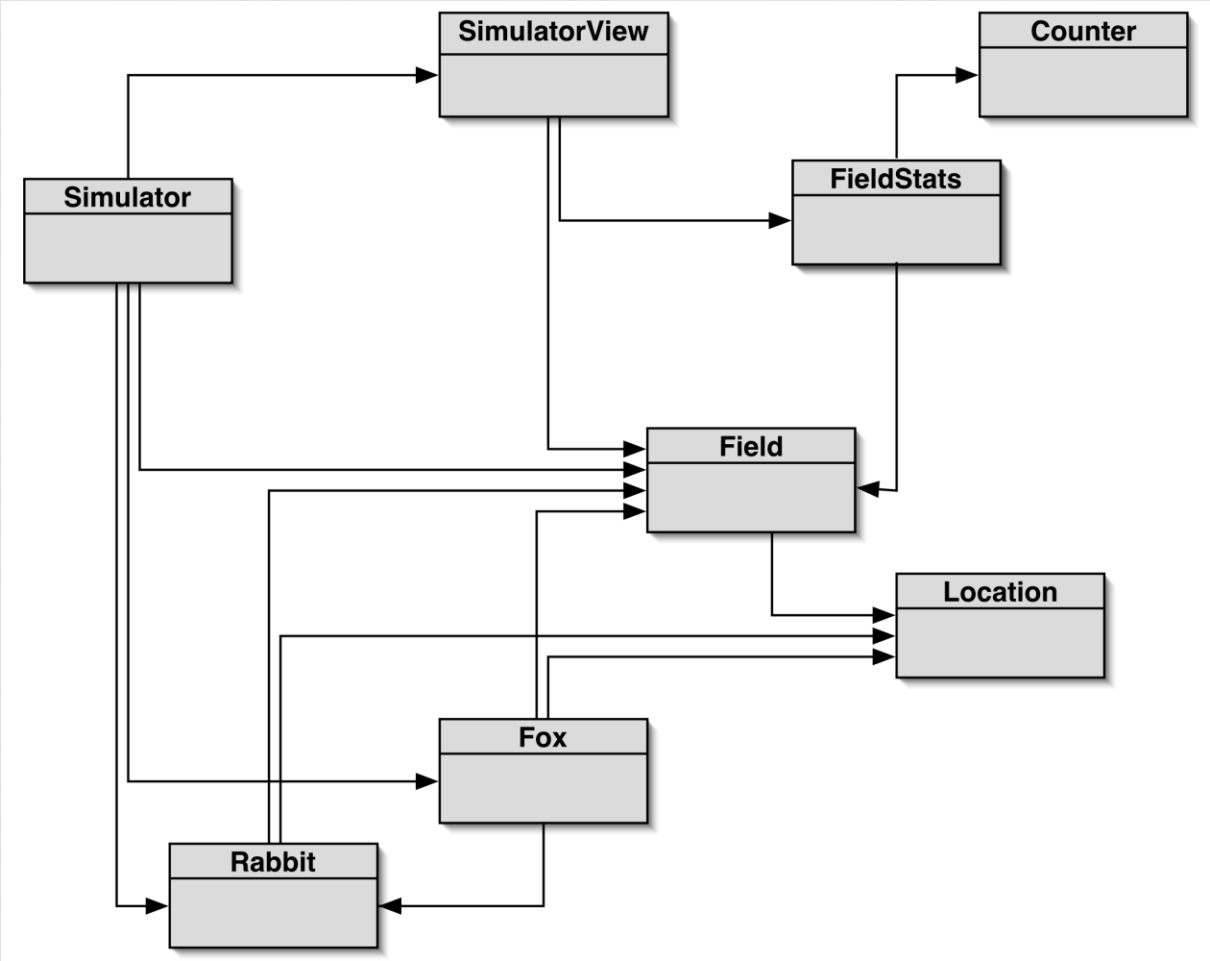
FUNCTIONEEL OF DATA GEORIËNTEERD

- functioneel georiënteerd :
  - structuur van het probleem ⇒
  - structuur van functies ⇒
  - structuur van het programma

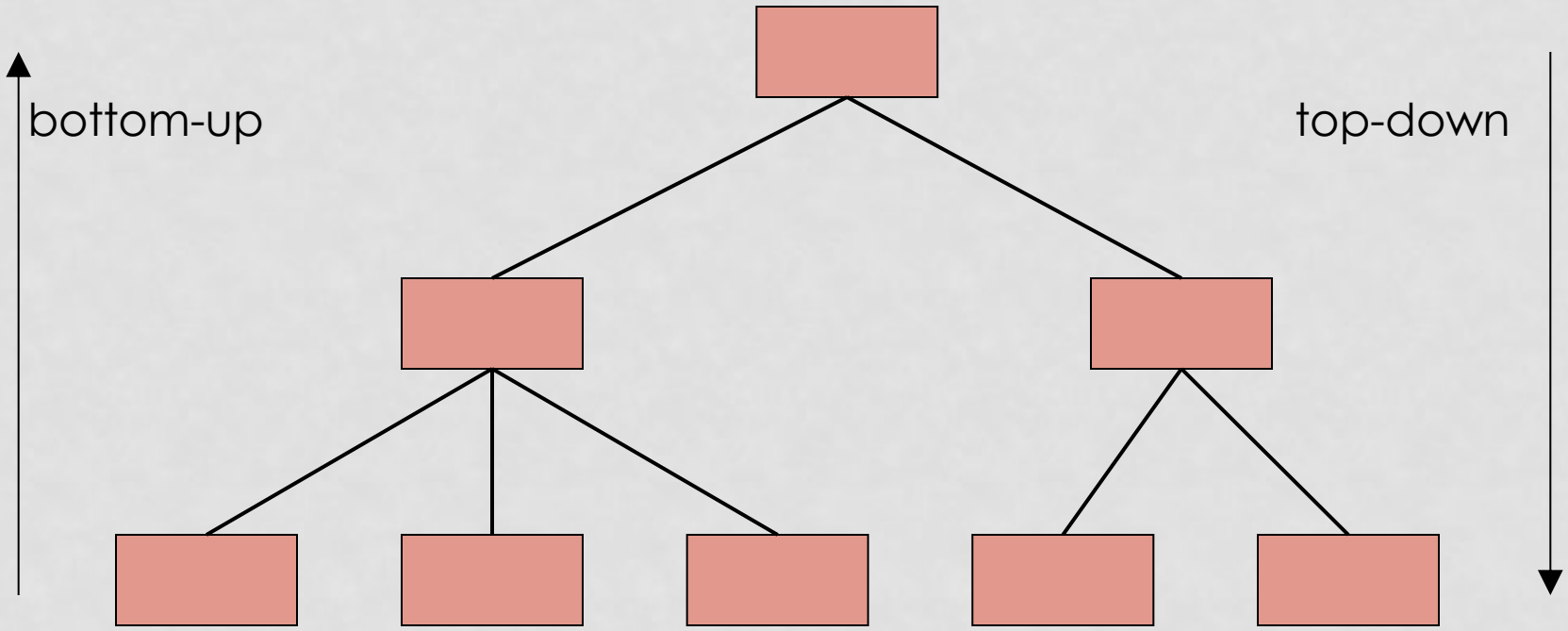
- data georiënteerd :
  - structuur van het probleem ⇒
  - structuur van de gegevens (data structuren) ⇒
  - structuur van het programma

8

OO = DATA GEORIËNTEERD



FUNCTIONAL DECOMPOSITION



10

FUNCTIES

```
// function prototype
int maximum (int, int);

int maximum (int x, int y)
{
    int max = x;

    .... statements ...

    return max;
}
```

declaratie functie

implementatie = block

FUNCTIE PROTOTYPE

- functie moet bekend zijn (in de file) voordat deze kan worden gebruikt (d.w.z. functie call)
- functie declaratie = functie prototype
- waarom ?
- forward declaratie: functie kan worden aangeroepen voordat deze is gedefinieerd
- compiler kan volgorde en type van variabelen en return waarde controleren
  - kan soms run-time error voorkomen
- functie beschikbaar maken voor andere files
  - denk aan library functies

file demo.c

```
#include <stdio.h>

f(x)                    <------ prototype ontbreekt
{
   return x;
}

int main()
{
   printf("%f\n", f(1.3));
}
```

```
$ gcc -W demo.c
demo.c: In function main:
demo.c:5: warning: format %f expects type double, but
   argument 2 has type int
demo.c: In function f:
demo.c:8: warning: type of x defaults to int

$ a.out
0.000000
```

DECLARATIE VS. DEFINITIE

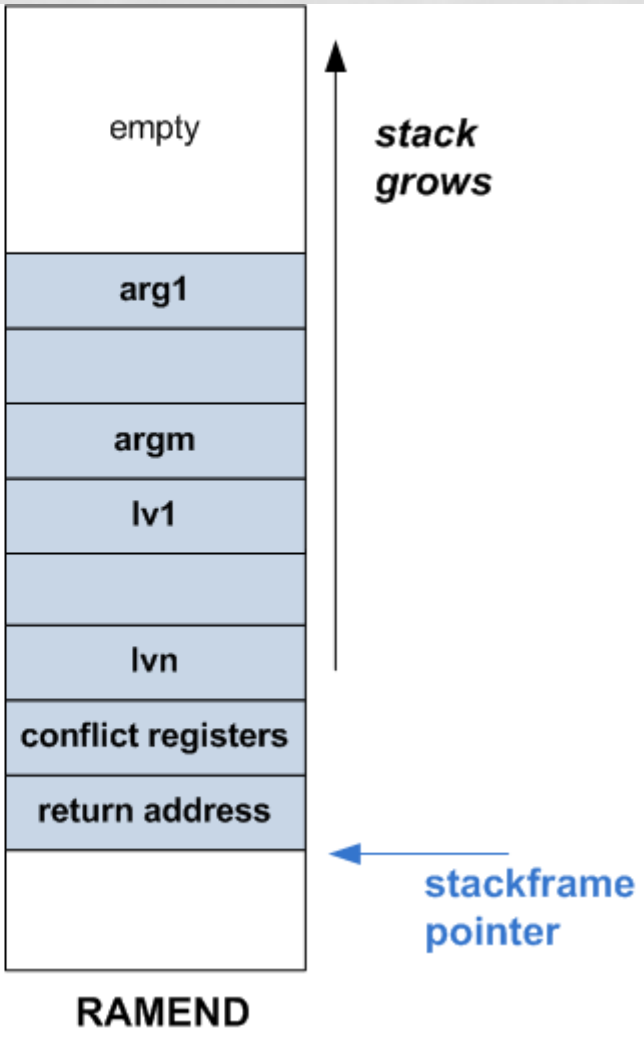
- in een programma mogen meerdere declaraties maar er mag maar één definitie zijn !
- declaratie
  - variabele/functie naam en type is nu bekend bij de compiler, geen allocatie van geheugen
  - variabele : int x;
  - functie prototype : int add(int, int);
- definitie
  - wijst geheugen toe
  - variabele : int x = 5;
  - bij functie : de body { ... }

GCC STACKFRAME

- elke functie-call geeft een stack frame op de stack
- stack frame heeft eigen pointer : stack frame pointer
- de stack frame wordt vrijgegeven bij return van functie
- 'stack smashing' : attack where stack buffer overflow is exploited
  - example : inject your code and overwrite return address

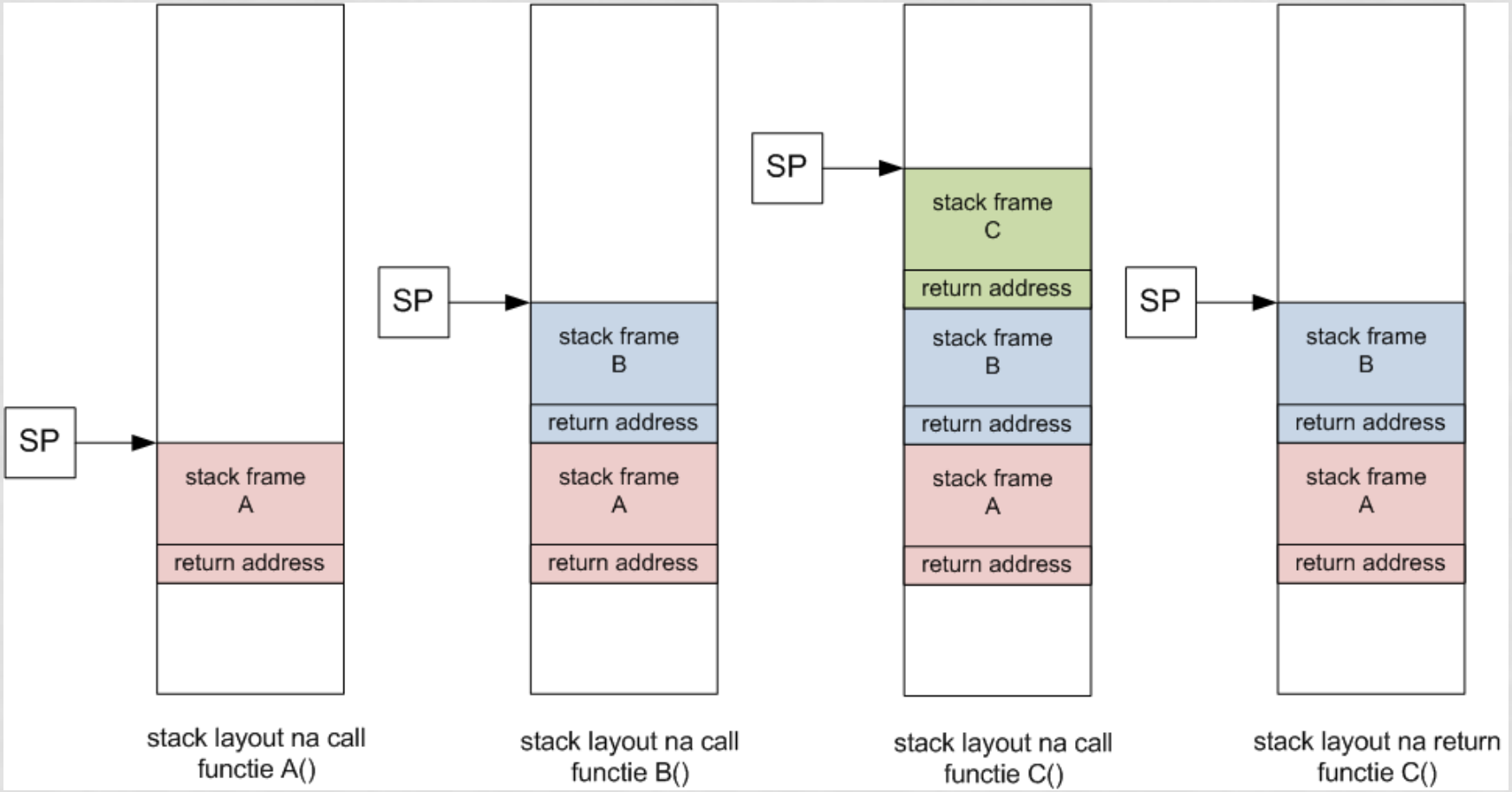
```
void f(arg1, arg2, …, argm)
{
   int lv1, lv2, …, lvn;

}
```



conflict registers :
– need to restore the old contents of these registers
– one conflict register is the stack frame pointer

GCC STACKFRAME



AGENDA

- control statements
- functies & stackframe
- **array & string**
- structure
- enum
- local & global scope

ARRAYS

- een array is een opeenvolgende structuur van hetzelfde type
- een aaneengesloten blok geheugen
- in C geen lengteveld, wel een functie sizeof()
- syntax is net even anders dan in Java
  - `int a[10]` : 10 integers a[0], a[1], ... a[9]
  - `uint8_t buffer[5]` : 5 bytes
  - `char arr[2] = {'a', 'b'}`
  - `int a[]` : only a pointer, no memory allocated

- Java : Byte[] myList;

STRINGS

- een string is een array van chars eindigend op '\0'
- er is geen type string in C
- de '\0' telt niet mee voor de lengte !

```
char s[] = "Let's go";
// string lengte 8, maar array lengte 9
```

IN C GEEN TYPE STRING

- `int a[] = {1,2,3,4,5};`
- `a = 56789;`

- `char s[10] = "rood";`
- `s = "blauw";`
- `strcpy(s, "blauw");`

STRINGS

- opdracht : maak een functie 'stringcopy()' die een string s kopieert naar string t

```
main ()
{
        char s[] = "Pietje Bell";
        char d[20];
        stringcopy (d, s);
        printf("%s\n", d);
}
```

```
void stringcopy(char[] des, char[] src)
{
      int i = 0;
      while (src[i] != '\0') {
        des[i] = src[i];
        i++;
      }
      des[i] = '\0';
}
```

AGENDA

- control statements
- functies & stackframe
- array & string
- **structure**
- enum
- local & global scope

STRUCTURES

- struct is een samenstelling van basistypes (of van andere structs)

```
struct [tag-name] {member-list} [var-name];
```

- voorbeeld :

```
struct point {int x; int y}; // declareer een type
struct point p; // declareer een variabele
```

- ook mag :

```
struct point {int x; int y} p;
```

- dan kan :

```
p.x=2;
p.y=3;
```

STRUCTURES

- in var declaratie moet je steeds 'struct' herhalen :

```
struct point {int x; int y}; // type sctruct point
struct point p = {0, 0}; // p heeft type struct point
```

- een array van 10 structs, r[i] is van het type point:

```
struct point r[10];
```

- nog een voorbeeld:

```
struct account {
    int account_number;
    char[20] first_name;
    char[20] last_name;
    float balance;
};
```

AGENDA

- control statements
- functies & stackframe
- array & string
- structure
- **enum**
- local & global scope

ENUM

- enum is een lijst van (constante) integer waarden

```
enum [tag-name] {enum-list} [var-name];
```

- voorbeelden :

```
enum boolean {FALSE, TRUE};
enum color {red, white, blue};
enum state {INIT, RUNNING, BLOCKED};
```

- in var declaratie moet je steeds 'enum' herhalen :

```
enum boolean b;
b = TRUE;
if (b) { ... }
```

ENUM

- telt vanaf 0 (tenzij anders aangegeven)
- vaak gebruikt bij state machines (FSM) :

```
enum {OFF=0, ON=1, STOP=2, GO=3} state;
```

AGENDA

- control statements
- functies & stackframe
- array & string
- structure
- enum
- **local & global scope**

SCOPE

- function prototype scope
  - parameters genoemd in functie prototype (deze zijn niet vereist)
  - hebben verder geen betekenis
- file scope
  - elke identifier gedeclareerd buiten een block
  - vanaf declaratie tot EOF
  - dat zijn : declaraties van functie prototypes en globale variabelen
- block scope
  - vanaf declaratie tot einde block (anders dan Java !!)
  - elke identifier gedeclareerd in een block { .. }

WAT WORDT AFGEDRUKT ?

```
int x = 5;
void a()
{
   int x = 8;
   printf("%d\n", x);
}

int main(void)
{
   printf("%d\n", x);
   int i, x = 0;
   for(i=0; i < 1; i++) {
        int x = 3;
        a();
        printf("%d\n", x);
   }
   return(0);
}
```

5, 8, 3

STATIC VARIABELEN

- een static variabele in een functie
  - heeft de levensduur van een globale variabele
  - behoudt zijn waarde tussen functie calls
  - scope van de variabele is nog steeds lokaal !
- globale variabelen en functie prototypes kunnen ook static zijn, maar dit betekent iets heel anders (naam wordt niet meegegeven aan linker)

```c
#include <stdio.h>

void foo()
{
    int a = 10;
    static int sa = 10;

    a += 5;
    sa += 5;

    printf("a = %d, sa = %d\n", a, sa);
}


int main()
{
    int i;

    for (i = 0; i < 10; ++i)
        foo();
}
```

```
a = 15, sa = 15
a = 15, sa = 20
a = 15, sa = 25
a = 15, sa = 30
a = 15, sa = 35
a = 15, sa = 40
a = 15, sa = 45
a = 15, sa = 50
a = 15, sa = 55
a = 15, sa = 60
```

```
113 uint8_t counting()
114 {
115                         /*0*/  /*1*/ /*2*/ /*3*/ /*4*/ /*5*/ /*6*/ /*7*/ /*8*/ /*9*/
116     uint8_t digits[] = { 0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f };
117
118     static uint8_t digit = 0;
119
120     sendCommand(0x40); // auto-increment address
121     write(strobe, LOW);
122     shiftOut(0xc0); // set starting address = 0
123     for(uint8_t position = 0; position < 8; position++)
124     {
125         shiftOut(digits[digit]);
126         _delay_ms(100);
127         shiftOut(0x00);
128     }
129
130     write(strobe, HIGH);
131
132     digit = (digit + 1) % 10;
133     return (digit == 0);
134 }
```