

Föreläsning 8

Tobias Wrigstad

*Pekare och arrayer. Dynamiska arrayer.
Pekararrayer och kommandorads-
argument.*



Helt OK program vid Lab 5&6

"Magiskt nummer"

```
struct db
{
    good_t goods[128];
    int size;
};

int main(void)
{
    struct db db;

    ...
    add_to_db(&db);
}
```

Minnesslöseri

Variabel deklarerad i main()



Rimliga förändringar inför Inlupp 2

Definition inte siffra

```
#define INITIAL_DB_CAPACITY 128
```

```
struct db  
{
```

```
    good_t *goods;
```

```
    int capacity;
```

```
    int size;
```

```
};
```

Pekare till en array på heapen
(kommer att vara ett träd!)

Ingen hårdkodad kapacitet

Lokal variabel

```
int main(void)  
{
```

```
    struct db db = { .capacity = INITIAL_DB_CAPACITY; }
```

```
    add_to_db(&db);
```

```
}
```

Skickas som parameter (jmf. PKD)



Helt OK program Inlupp 1

lager.c

```
struct list {
    struct link *first, *last;
};

struct link {
    int value;
    struct link *next;
};

typedef struct list list_t;

int length(list_t*);
int empty(list_t*);

struct link* mk_link(...);

void append(...) {
    ...
}

int length(...) {
    ...
}

int empty(...) {
    ...
}
```



Rimliga förändringar inför Inlupp 2

lager.h

```
#ifdef
#define

typedefs...

functionprototyper...

#endif
```

lager.c

```
#include "lager.h"

strukturedefinitioner...

funktionsprototyper...

implementation av koden
```

...plus flera andra



Kraven växer med din kunskap!

- Första inluppen: vi fokuserar på att programmet gör **vad** det skall

Komma igång med C (se Z100)

- Andra inluppen: vi fokuserar på **hur** programmet gör det den skall

Minneshantering, globala variabler, datastrukturer, moduluppdelning, namngivning, etc. (se Z101)

Samt programmet i dess sammanhang: testning, makefiler, kodgranskning.

Pekare och arrayer (är nästan samma sak)

- Vad är skillnaden mellan dessa?

```
char *s = "Hello";
```

```
char s[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

```
char s[] = "Hello";
```

- De sista två är exakt samma, strängarna hamnar på stacken. Den första lägger strängen i "programmet" (ROM), eftersom den pekar på "Hello" som ligger i programmet.

```
s[0] == 'H'
```

```
s[5] == '\n'
```

```
s[6] == vad?
```

Pekare och arrayer (är nästan samma sak)

- Vad är skillnaden mellan dessa?

```
char *s = "Hello";
```

```
char s[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

```
char s[] = "Hello";
```

```
char *s = strdup("Hello");
```

```
char s[] = strdup("Hello"); // kompilerar ej
```



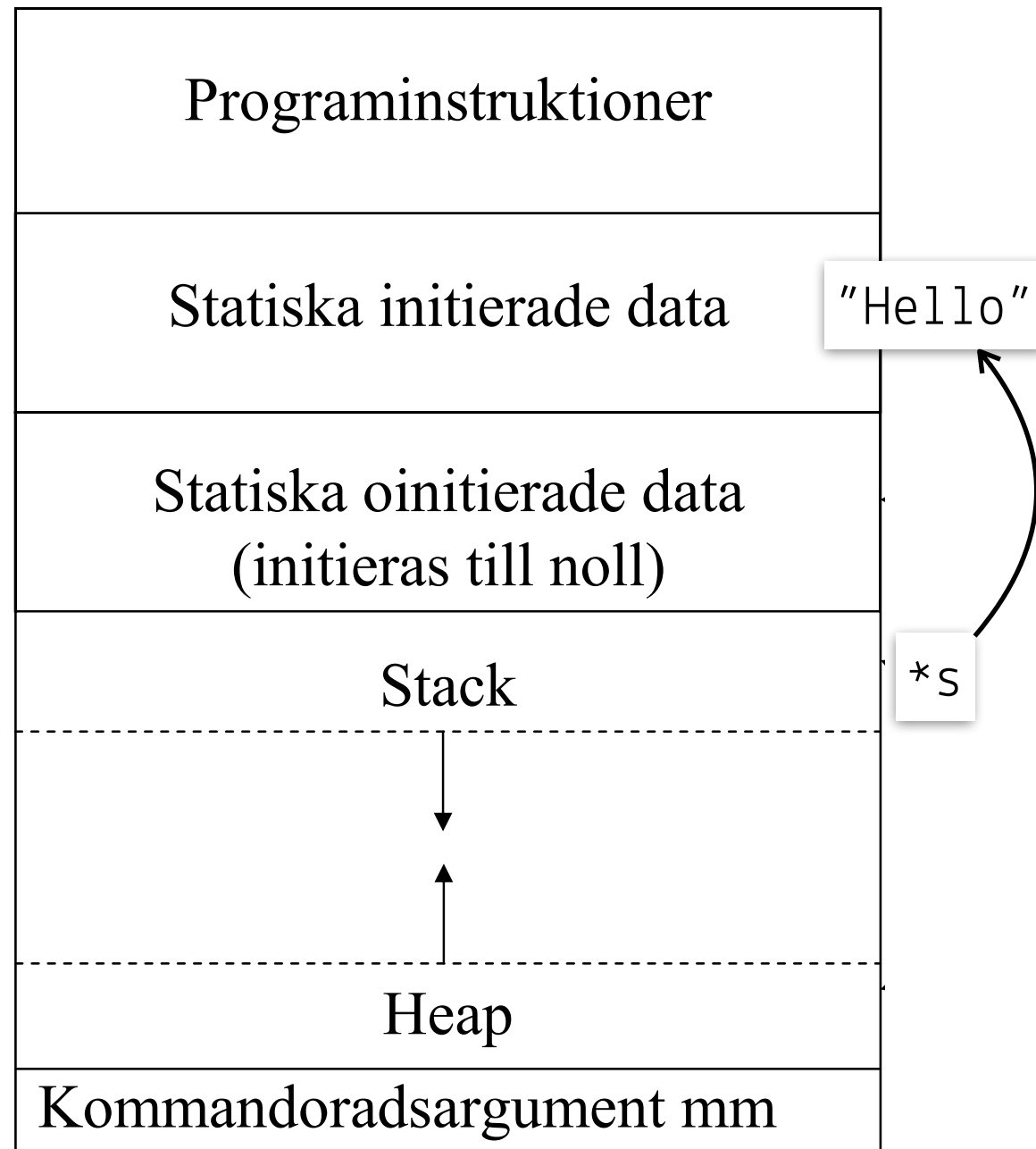
```
#include <string.h>
```

```
int main(int argc, char *argv[])  
{  
    char *s1 = strdup("Hello"); // 5  
    char s2[] = strdup("Hello"); // 6  
    return 0;  
}
```

```
$ gcc temp.c  
foo.c:6:8: error: array initializer must be an  
initializer list or string literal  
    char s2[] = strdup("Hello");  
              ^
```



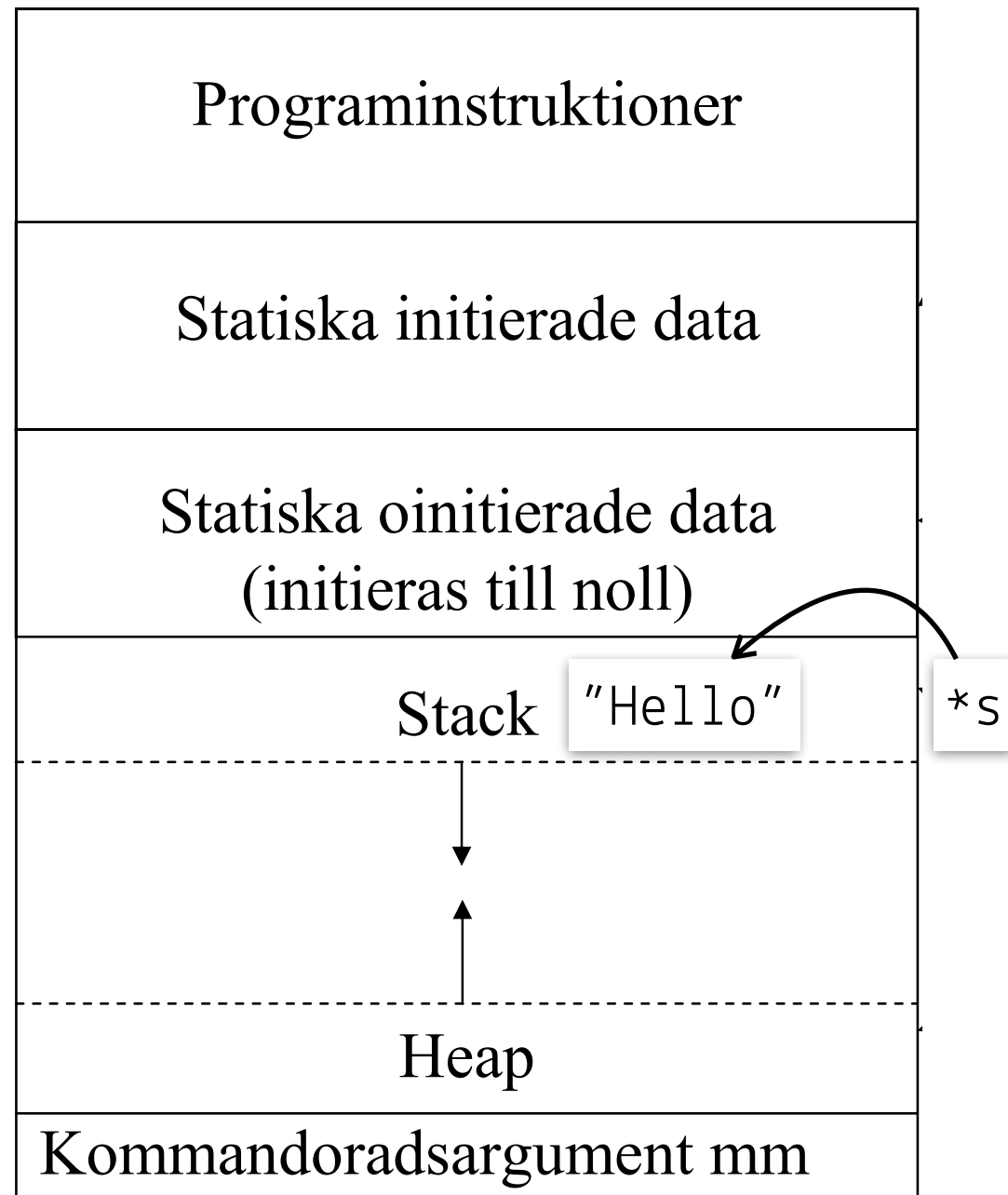
```
char *s = "Hello";
```



```
s[4] = 'x'; // BOOM!
```

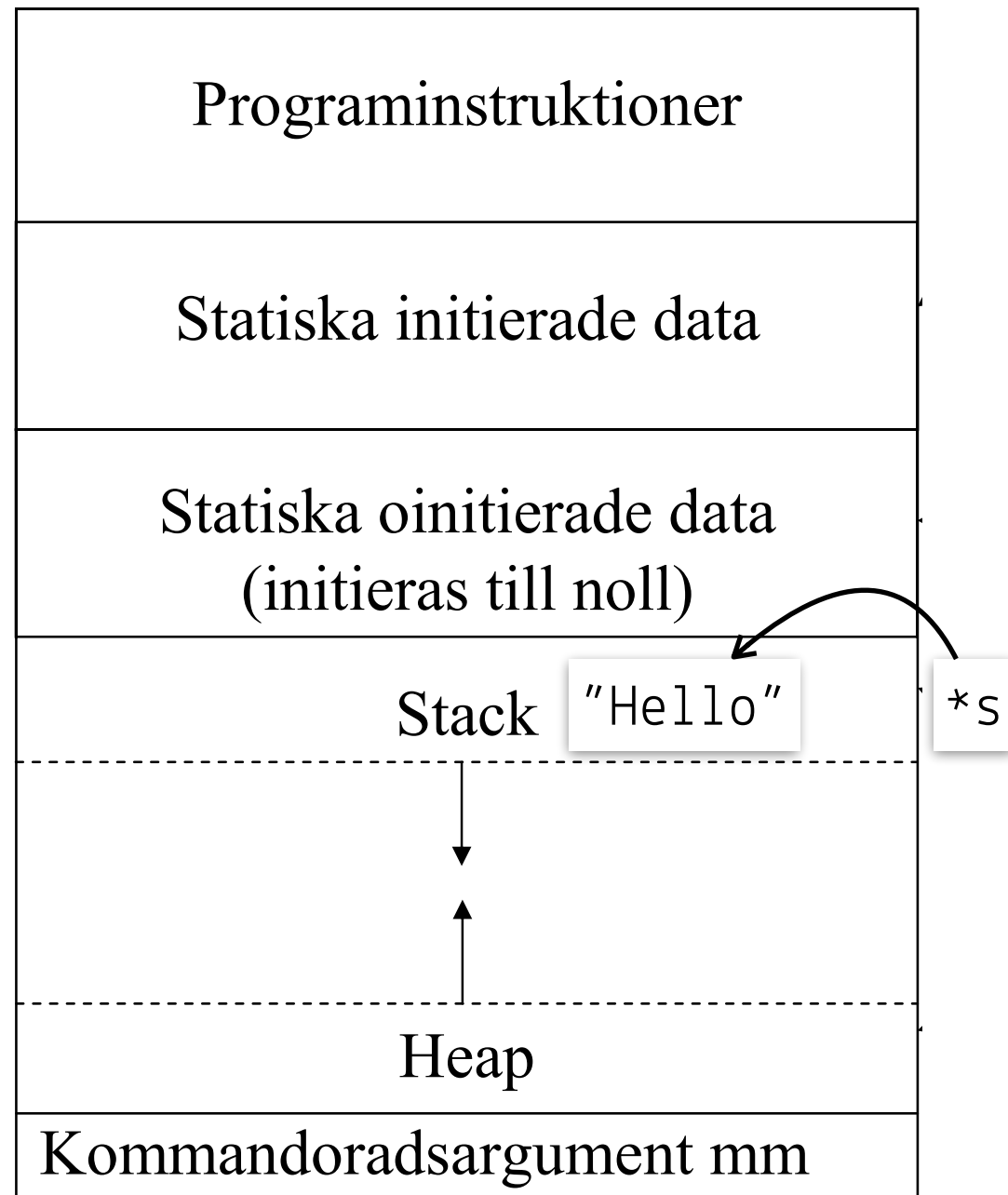
```
s[6] = ...
```

```
char s[] = "Hello";
```



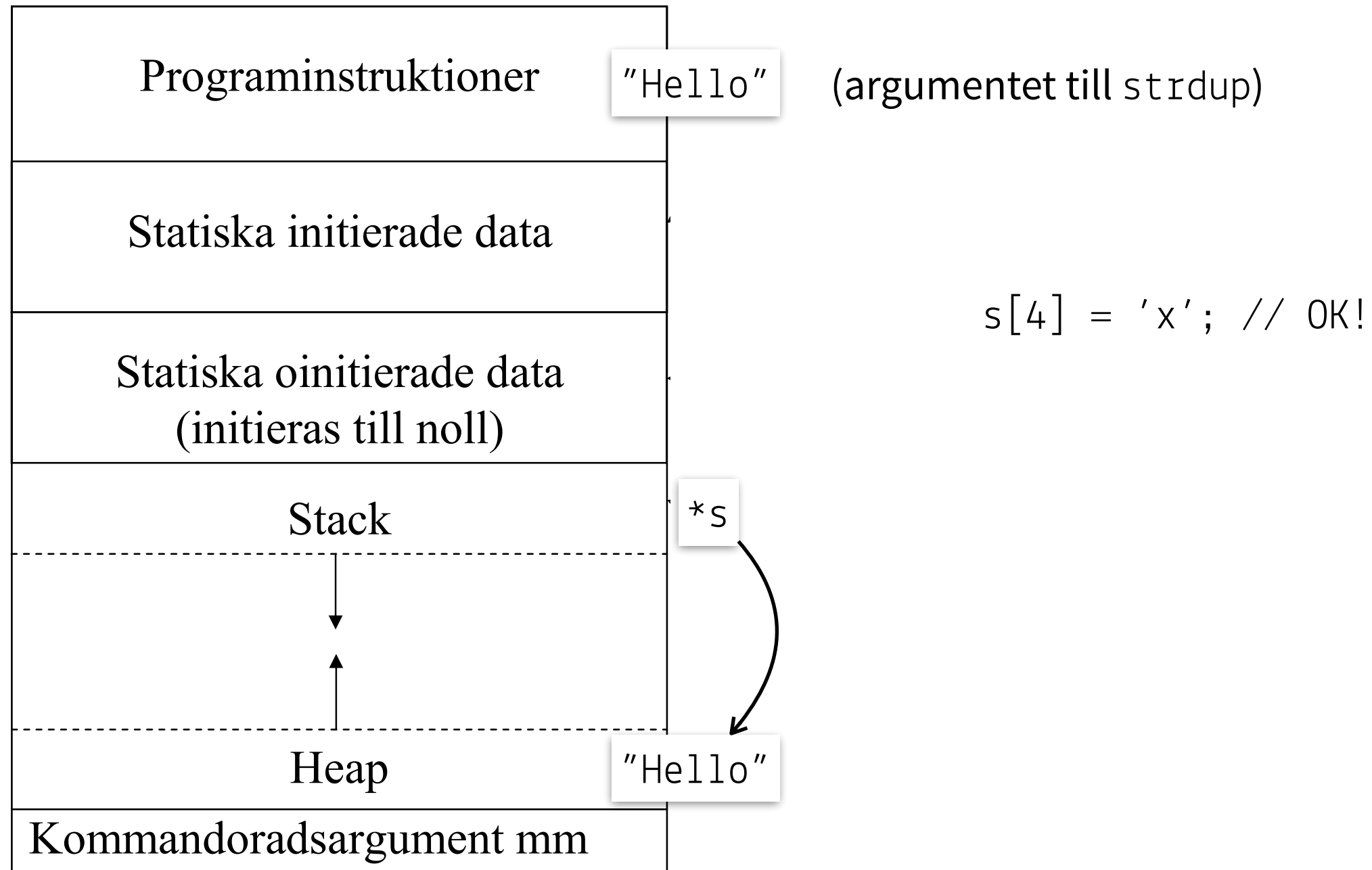
```
s[4] = 'x'; // OK!
```

```
char s[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```



```
s[4] = 'x'; // OK!
```

```
char *s = strdup("Hello");
```



Vanlig felkälla i lagerhanteraren

- Läsa in en sträng:

```
char buf[256];  
scanf("%s", buf);  
return buf;
```

- Vad är felet?
- Hur skiljer sig detta (också felaktiga) program?

```
char *buf;  
scanf("%s", buf);  
return buf;
```

Två lösningar

- Läs in i buffert, kopiera strängen på heapen, returnera pekare till kopian

```
char buf[256];  
scanf("%s", buf);  
return strdup(buf);
```

- OBS! Kräver att strängen frigörs med `free` på annan plats i programmet!
- Ännu bättre lösning (varför):

```
char *buf = NULL;  
size_t buf_len = 0;  
getline(&buf, &buf_len, stdin);  
return buf;
```

- *(Man kan också skicka med en buffert från anropsplatsen.)*

STACK

HEAP

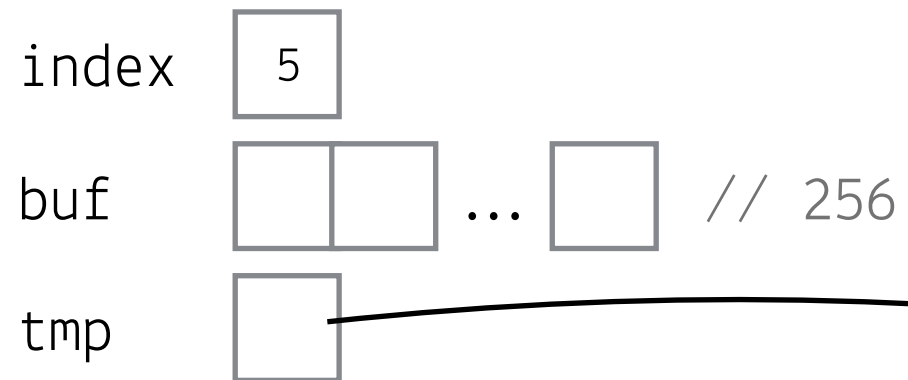
STATISKA INITIERADE VARIABLER

PROGRAMINSTRUKTIONER

```
void set_name(int index)
{
    char buf[256];
    scanf("%s", buf);
    char *tmp = strdup(buf);
    DB.goods[index].name = tmp;
}
```



STACK



HEAP



PROGRAMINSTRUKTIONER

```
void set_name(int index)
{
    char buf[256];
    scanf("%s", buf);
    char *tmp = strdup(buf);
    DB.goods[index].name = tmp;
}
```

STATISKA INITIERADE VARIABLER

DB



Använd alltid funktioner ”som terminerar”!

- Många standardfunktioner har en version som också tar ett gränsvärde:

`strncpy` — jämför de första n tecknen i två strängar (terminerar efter n steg)

`strncpy` — kopiera n tecken från a till b (terminerar efter n steg)

`getline` — allokerar själv en buffert som rymmer indata

...

- Försök från och med nu att undvika kod som ser ut så här:

```
char buf[256];
```

```
scanf("%s", buf); // kraschar om input är större än 256
```

```
return strdup(buf);
```

- Observera att lösningen **inte** är ”en större buffert”.

Dynamiska arrayer

- Exempel

Hur kan man implementera en array i C som kan växa och krympa?

- Exemplifierar

Manuell minneshantering

Värdesemantik vs. pekarsemantik

```
#include <string.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>

typedef int T;

struct dyn_array
{
    uint16_t capacity; // 64K element max
    uint16_t used;
    T *elements;
};

typedef struct dyn_array dyn_array_t;
```

Interface

`darray_create` — skapa en ny dynamisk array med en given kapacitet

Allokera minne!

`darray_free` — frigör en dynamisk array

Avallokera minne!

`darray_set` — uppdatera ett givet element med indexkontroll

`darray_get` — skaffa en pekare till ett givet element med indexkontroll

`darray_append` — öka storleken på arrayen och lägg till ett nytt element sist

Ändra på minnesstorlek!

`darray_prepend` — öka storleken på arrayen och lägg till ett nytt element först

Ändra på minnesstorlek!

```
// i main  
a = darray_create(4);
```

darray_create

capacity

4

capacity

4

elements

main

a

capacity

4

elements

Stack

Heap

0

0

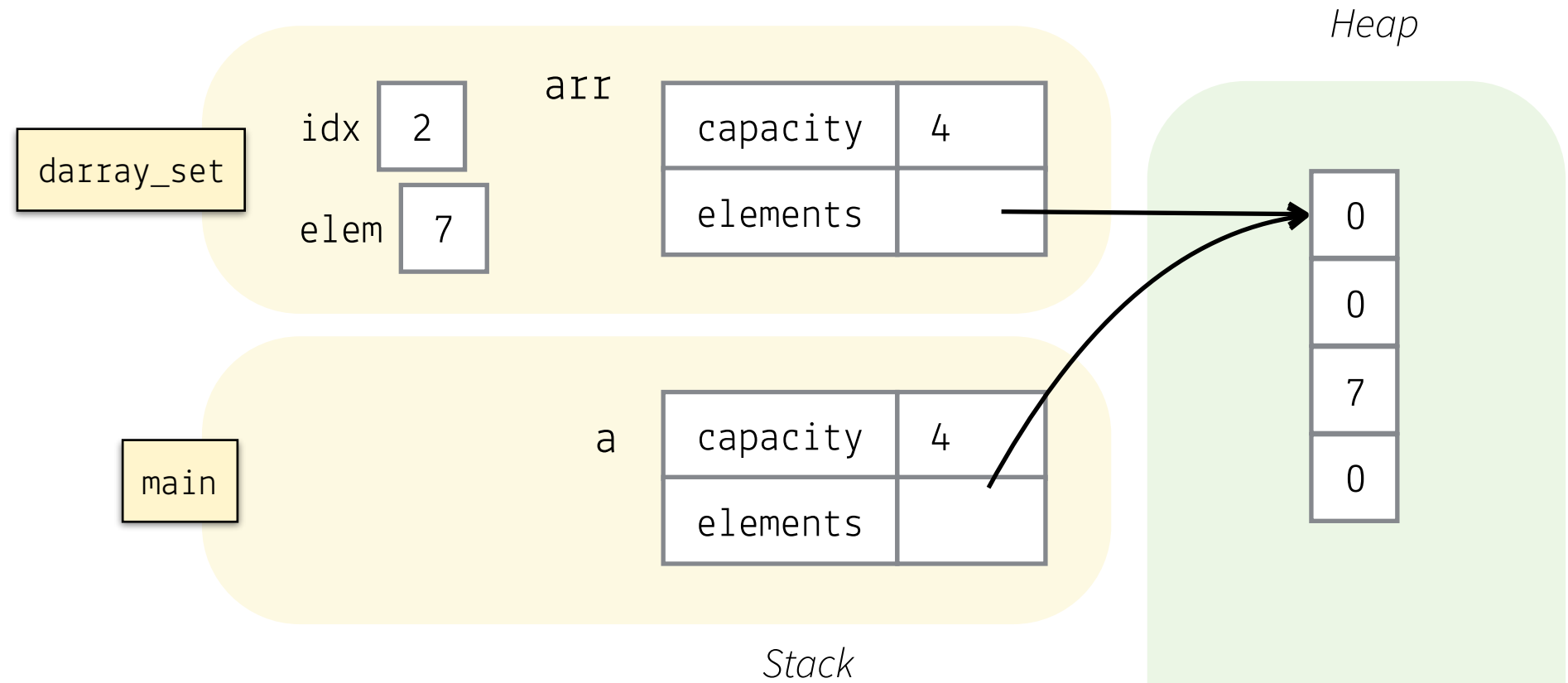
0

0

```
dyn_array_t darray_create(uint16_t capacity)  
{  
    // OBS! Borde göra felkontroll!  
    return (dyn_array_t) {  
        .capacity = capacity,  
        .elements = calloc(capacity, sizeof(T)) };  
}  
  
void darray_free(dyn_array_t *arr)  
{  
    free(arr->elements);  
    free(arr);  
}
```



darray_set(a, 2, 7);

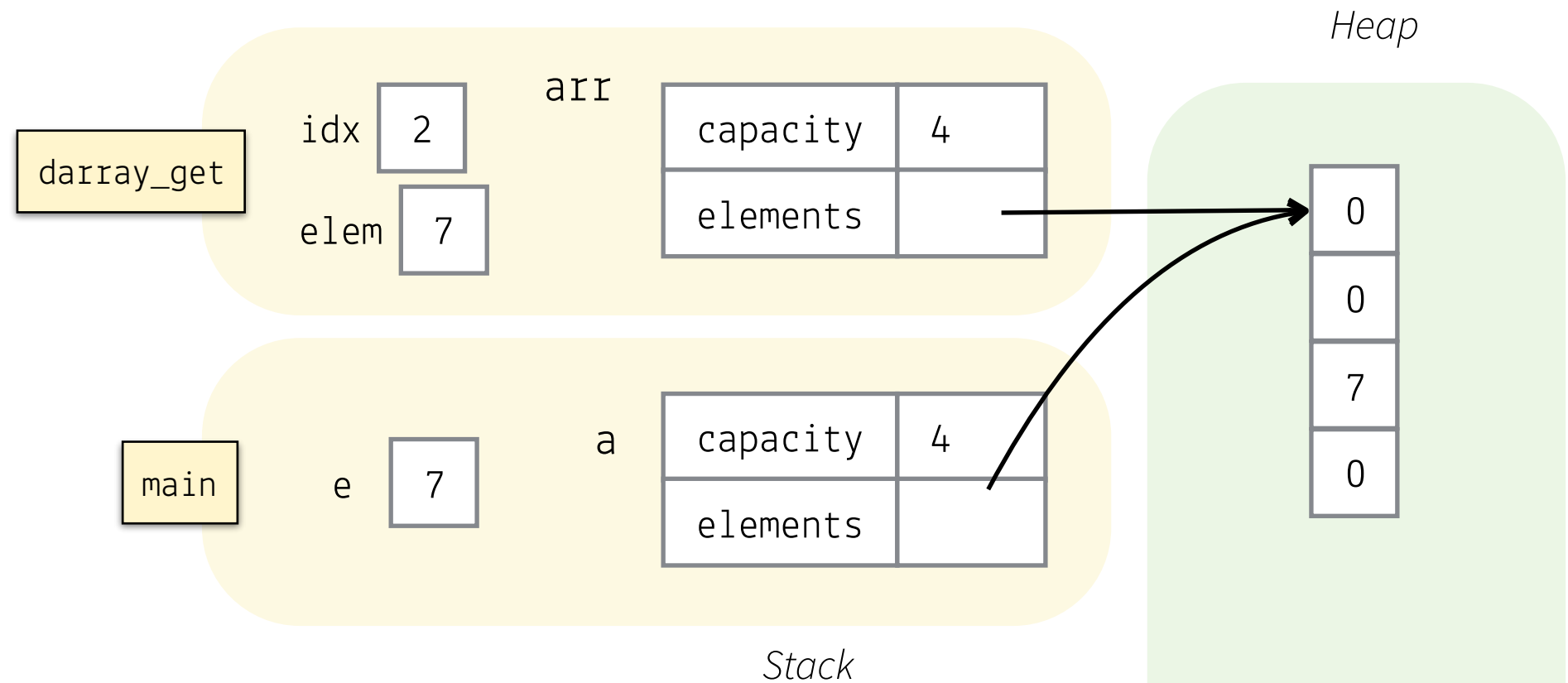


```
bool darray_set(dyn_array_t arr, uint16_t idx, T elem)
{
    if (idx < arr.capacity)
    {
        arr.elements[idx] = elem;
        return true;
    }

    return false;
}
```



`e = *darray_get(a, 2);`



```
T *darray_get(dyn_array_t arr, uint16_t idx)
{
    return (idx < arr.capacity) ? &arr.elements[idx] : NULL;
}
```

```
bool darray_get(dyn_array_t arr, uint16_t idx, T *result)
{
    ... // övning!
}
```



darray_append(&a, 9);

darray_append

elem 9

arr

main

a

capacity	8
elements	

Stack

Heap

0
0
7
0
9
0
0
0

```
void darray_append(dyn_array_t *arr, T elem)
{
    int idx = arr->capacity;

    arr->capacity *= 2;
    arr->elements =
        realloc(arr->elements, arr->capacity * sizeof(T));

    arr->elements[arr->idx] = elem;
}
```



darray_prepend(&a, 1);

darray_prepend

elem

9

arr

main

a

capacity

8

elements

Stack

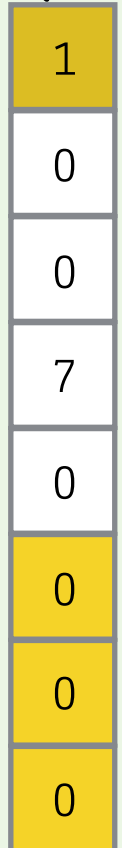
Heap

```
void darray_prepend(dyn_array_t *arr, T elem)
{
    int max = arr->capacity;

    arr->capacity *= 2;
    arr->elements =
        realloc(arr->elements, arr->capacity * sizeof(T));

    for (int i = max; i > 0; --i)
        arr->elements[i] = arr->elements[i-1];

    arr->elements[0] = elem;
}
```



realloc och calloc

- `ptr = realloc(ptr, new_size)`

Ändrar storleken på ett minnesutrymme, möjligen genom att flytta det

Farligt om det finns alias till `ptr`

- `ptr = calloc(number, size)`

Allokerar `number * size` antal bytes

Nollställer minnet

Frivillig övningsuppgift hemma

- Varför används pekarsemantik ibland och värdesemantik ibland?

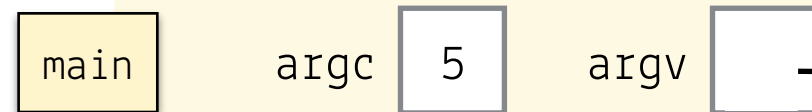
Vad skulle hända om man bytte från pekarsemantik till värdesemantik eller tvärtom i t.ex. `darray_prepend`?

- Hur fungerar `malloc`, `free`, `calloc` och `realloc`?

Läs gärna man-sidorna (`$ man calloc`) så du har koll på man till kodprovet!

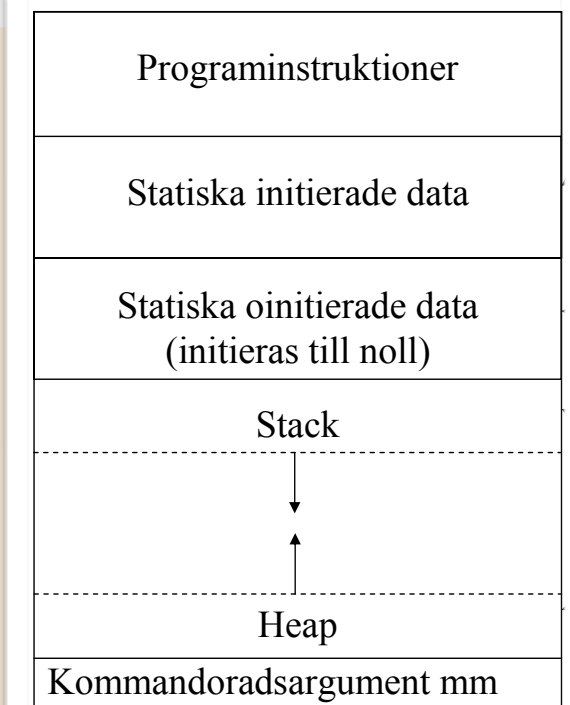
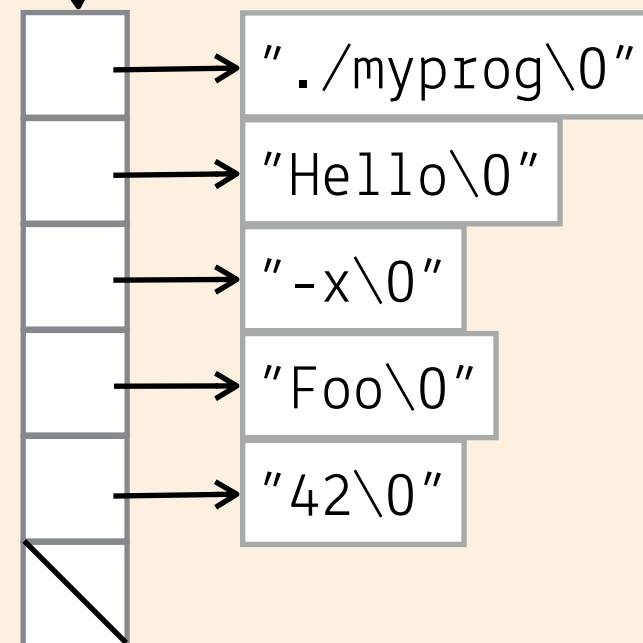
- Om man ändrade på typen `T` till att vara en pekare — vad skulle hända då med biblioteket?

Pekararrayer och kommandoradsargument



```
int main(int argc, char *argv[])
{
    while (*argv) puts(*argv++);
    return 0;
}
```

```
$ ./myprog Hello -x Foo 42
```



Läsbarhet?

```
int main(int argc, char *argv[])
{
    while (*argv) puts(*argv++);
    return 0;
}
```

```
int main(int argc, char *argv[])
{
    for (int i = 0; i < argc; ++i)
    {
        puts(argv[i]);
    }

    return 0;
}
```

Förstör inte heller argv!

Genericitet

- Vår dynamiska array tog emot en pekare av typen `T` som var definierad som en `int`

Återanvändning — man kan ändra `T` till något annat och kompilera om

Återanvändning flera gånger i samma program?

Två möjligheter: skapa ett makro som skapar flera datastrukturer — eller `void *`

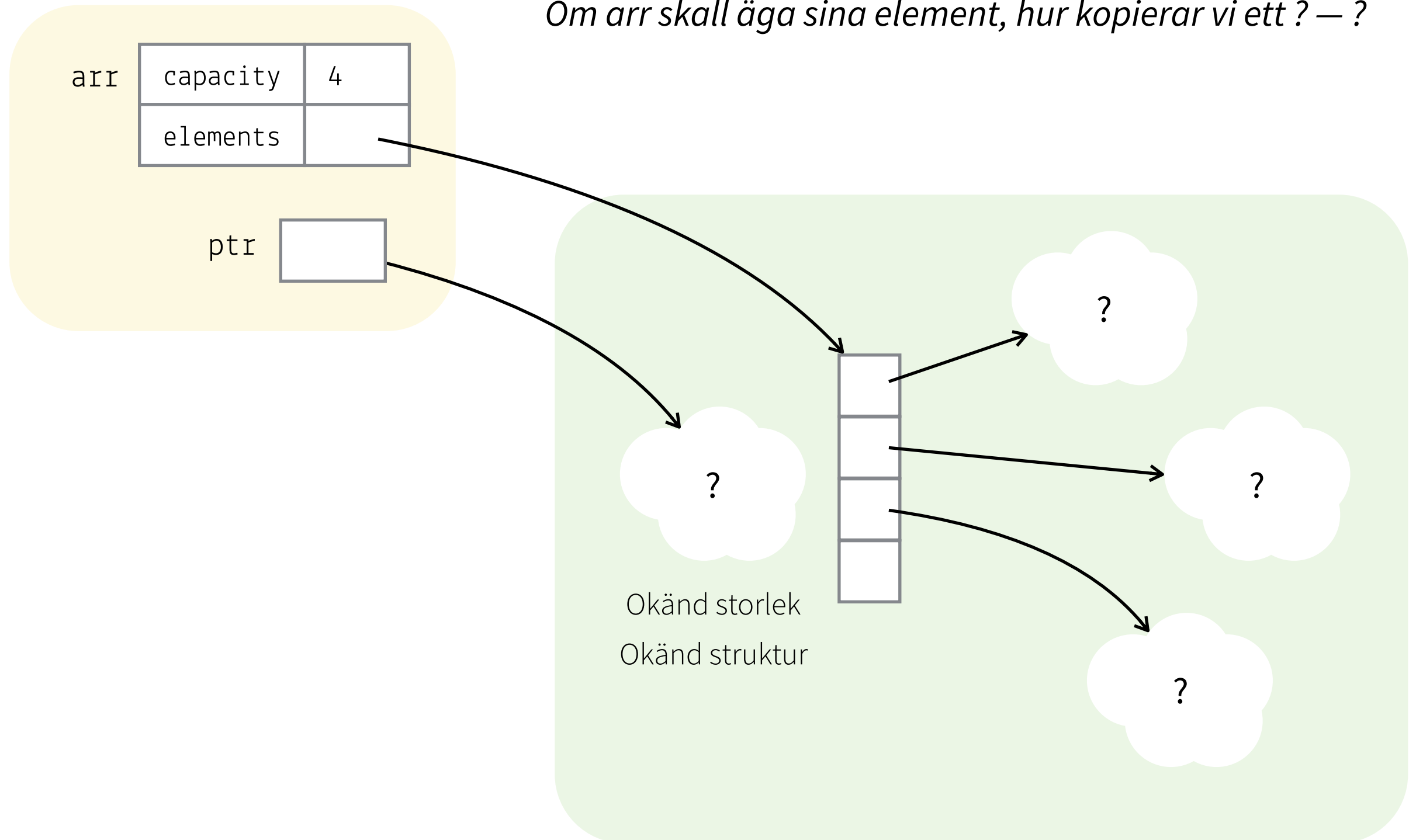
- Använda **`typedef void *T;`**

Eventuellt problem: kan inte längre beräkna `sizeof(*T)` (— varför inte?)

- Scenario: vi vill att den dynamiska arrayen skall äga sitt minne

darray_set(arr, 3, ptr)

Om arr skall äga sina element, hur kopierar vi ett? — ?



...men vad händer om T innehåller pekare?

```
void darray_free(dyn_array_t *arr)
{
    for (int i = 0; i < arr->capacity; ++i)
    {
        free(arr->elements[i]); // kan läcka minne!
    }
    free(arr->elements);
    free(arr);
}
```

Lösning: använd en funktionspekare!

Generella datastrukturer och funktionspekare

void *



En generell datastruktur

- Orimligt att ha en separat kodbas för varje listtyp (e.g., `intlist_t`, `foolist_t` etc.)
- Vi vill kunna skriva en lista som fungerar för alla datatyper

Problem: om vi inte vet vilken datatyp som skall lagras i listan är det inte möjligt för kompilatorn att räkna ut storleken för varje strukt

```
typedef struct node node_t;
struct node
{
    node_t next;
    T element; // vad är t?
};
```

Lösning: Pekare

- En pekare har alltid samma storlek oavsett vad som pekas ut
- C stöder (nästan) polymorfism för pekare via s.k. **void**-pekare, **void ***

Problem: vi kan inte operera på data av typen **void ***, eftersom vi inte vet något om den

```
typedef struct node node_t;
struct node
{
    node_t next;
    void *element; // känd storlek
};
```

Lösning: Funktionspekare

- En pekare till en funktion som kan anropas via pekaren
- Kompilatorn

```
int strcmp(char *a, char *b)
{
    ...
}

strcmp; // pekare till funktionen
```

Exempel

```
typedef struct list list_t;
typedef struct node node_t;
```

```
struct node
{
    node_t *next;
    void *elem;
};
```

```
struct list
{
    node_t *first;
    comparator cmp;
};
```

```
list_t *list_new(comparator cmp)
{
    list_t *result = malloc(sizeof(list_t));
    *result = (list_t) { .cmp = cmp, .first = node_new(NULL, NULL) };
    return result;
}
```

```
void list_insert(list_t *list, void *elem)
{
    node_t *p = list->first;
    for (node_t *n = p->next; n; p = n, n = n->next)
    {
        if (list->cmp(elem, n->elem) < 0)
        {
            p->next = node_new(elem, n);
            return;
        }
    }
    p->next = node_new(elem, NULL);
}
```



Vad är en comparator?

- **typedef int**(*comparator)(**void** *, **void** *);

Bakvänd syntax

Definierar typen `comparator` som en pekare till en funktion som tar emot två **void**-pekare och returnerar en **int**

Alla funktioner som tar två pekare och returnerar något som ryms i en `int` matchar denna typ, t.ex. `strcmp`

```
· list_t *l = list_new(strcmp); // En sorterad lista av strängar!  
  list_insert(l, "foo");  
  list_insert(l, "bar");
```

Nu kan vi definiera en "int-lista"

- ```
int intcmp(void *a, void *b) {
 if (*a == *b) { return 0; } else
 if (*a < *b) { return -1; } else
 { return 1; }
}
```
- ```
list_t *l = list_new(intcmp); // En sorterad lista med intpekare!  
int a = 5;  
int b = 7;  
list_insert(l, &a);  
list_insert(l, &b);
```


Fler funktionspekare

- I listexemplet kan det vara motiverat med en pekare till en funktion som vet hur man tar bort (free) element i listan

```
struct list
{
    node_t *first;
    comparator_f cmp;
    free_f free;
};
```

```
typedef void (*free_f)(void *);
```

```
void free_stuff(stuff_t *s)
{
    free(s->foo);
    free(s);
}
```



Klarar av att ta bort länkade strukturer