Föreläsning 7

Tobias Wrigstad

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

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).

```
s[0] == ’H’

s[5] == ’\n’

s[6] == vad?
```

Pekare och arrayer (är nästan samma sak)

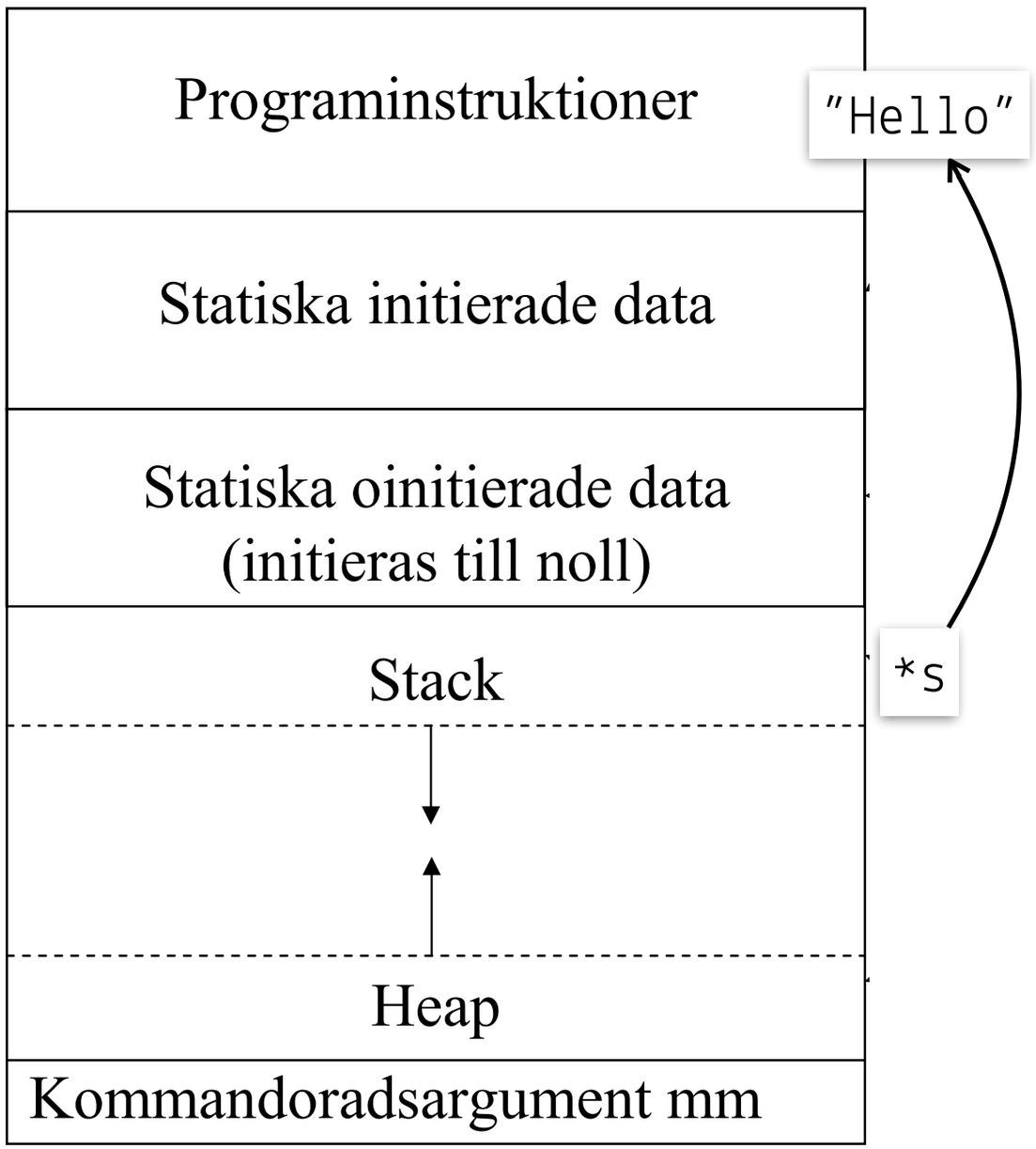
- Vad är skillnaden mellan dessa?

```
char *s = "Hello";

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

char s[] = "Hello";

char *s = strdup("Hello");

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

```c
#include <string.h>

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

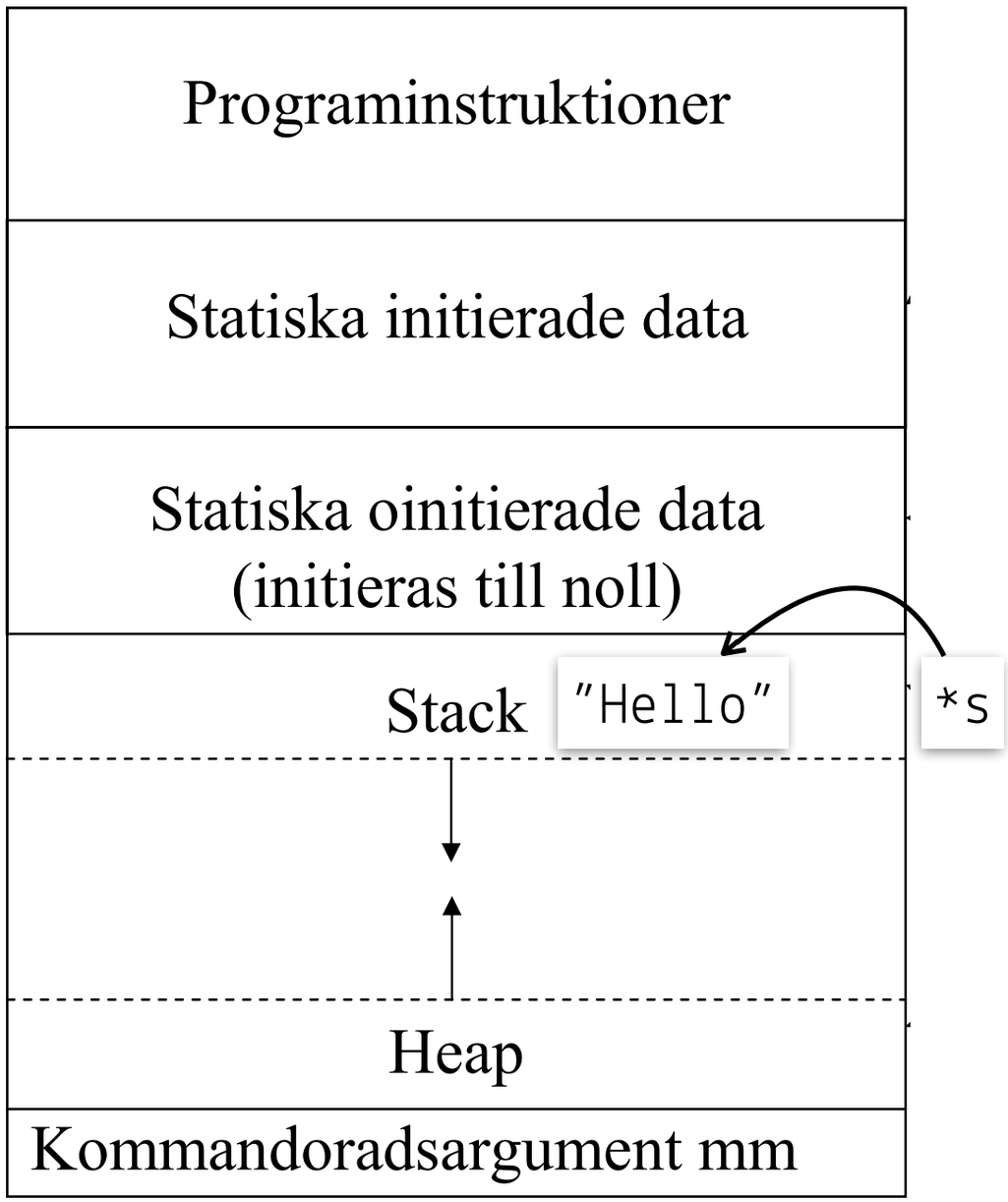
```
$ gcc temp.c
temp.c:6:8: error: redefinition of 's' with a different
type: 'char []' vs 'char *'
  char s[] = strdup("Hello");
       ^
temp.c:5:9: note: previous definition is here
  char *s = strdup("Hello");
```

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



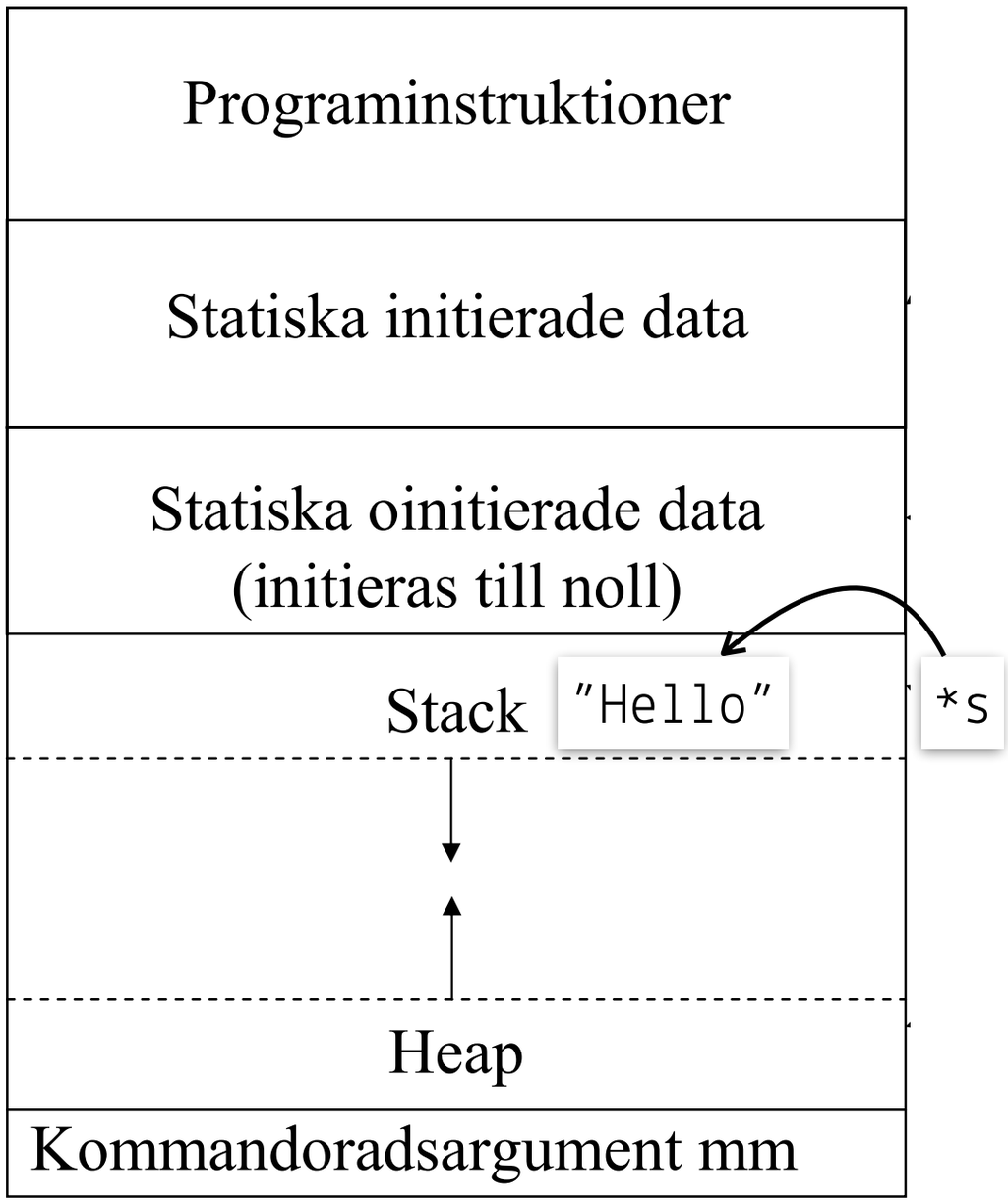

```
s[4] = 'x'; // BOOM!

s[6] = ...
```

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



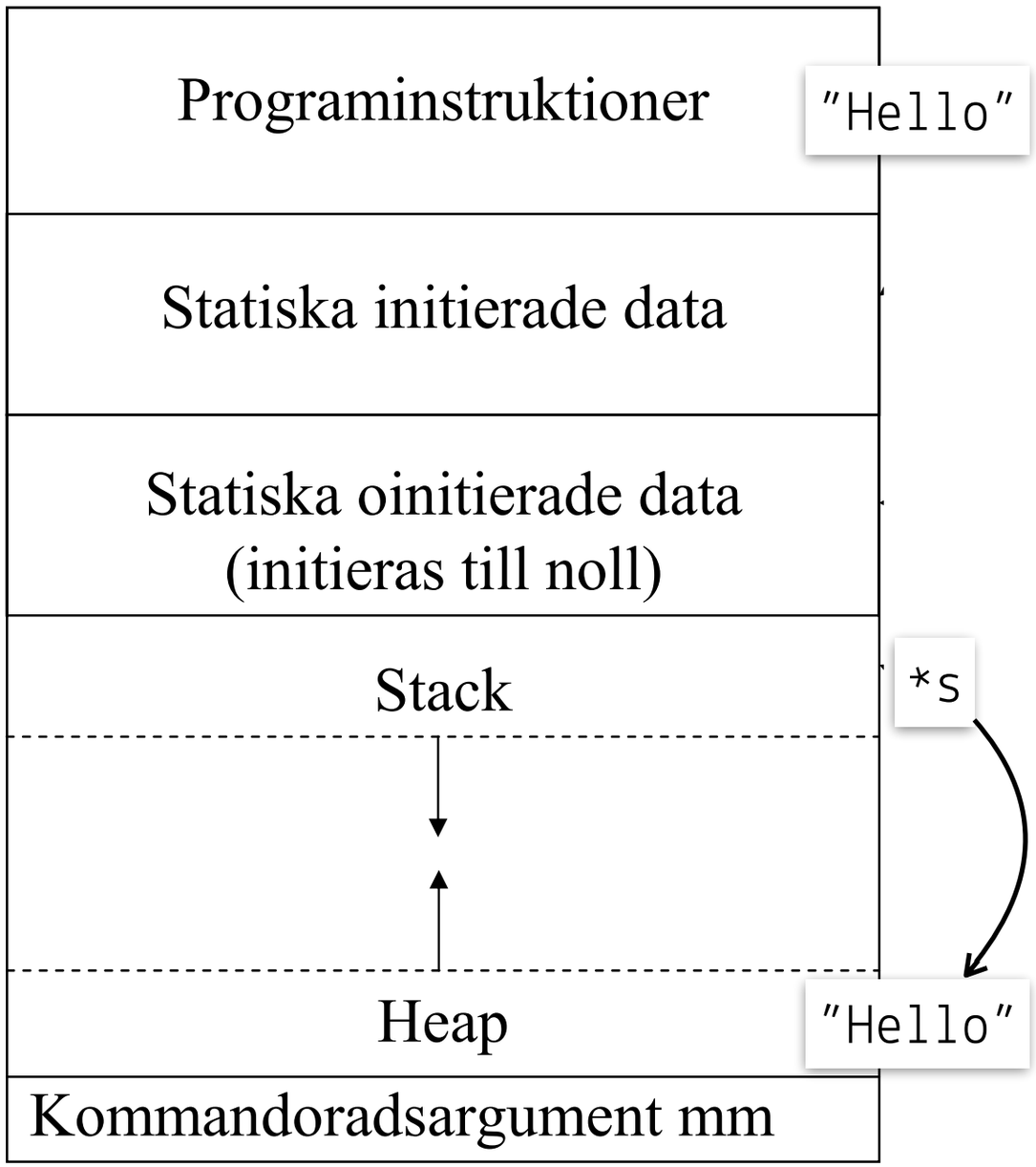

```
s[4] = 'x'; // OK!
```

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




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

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



(argumentet till `strdup`)

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

Vanlig felkälla i lagerhanteraren

- Läsa in en sträng:

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

- Vad är felet?

- Hur skiljer sig detta 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;
size_t buf_len;
getline(&buf, &buf_len, stdin);
return buf;
```

Använd alltid funktioner ”som terminerar”!

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

  `strncmp` — jämför de första *n* tecknen i två strängar (terminerar efter *n* steg)
  `stncpy` — 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

```c
#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

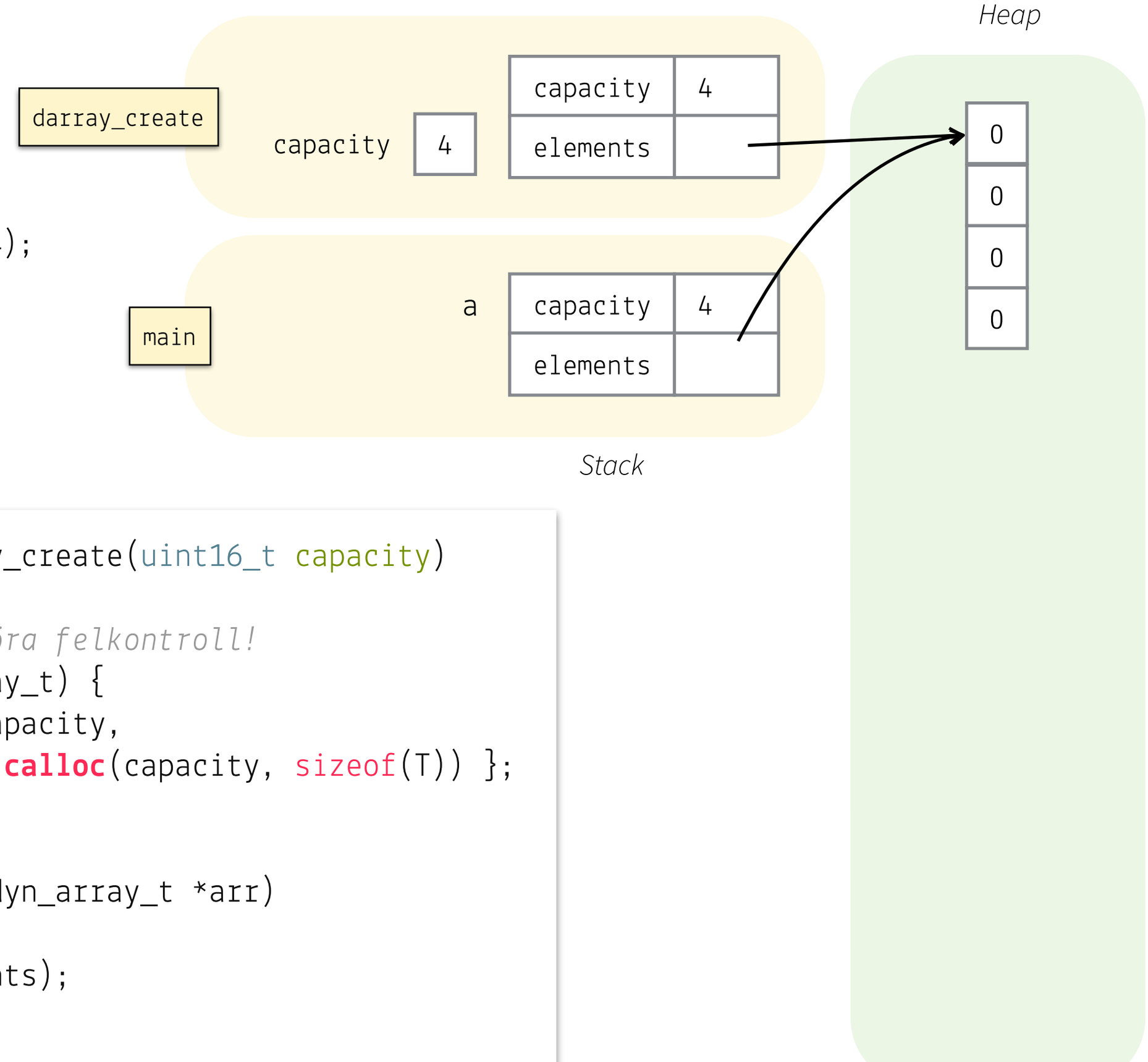
`darray_append` — öka storleken på arrayen och lägg till ett nytt element sist

Ändra på minnesstorlek!

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

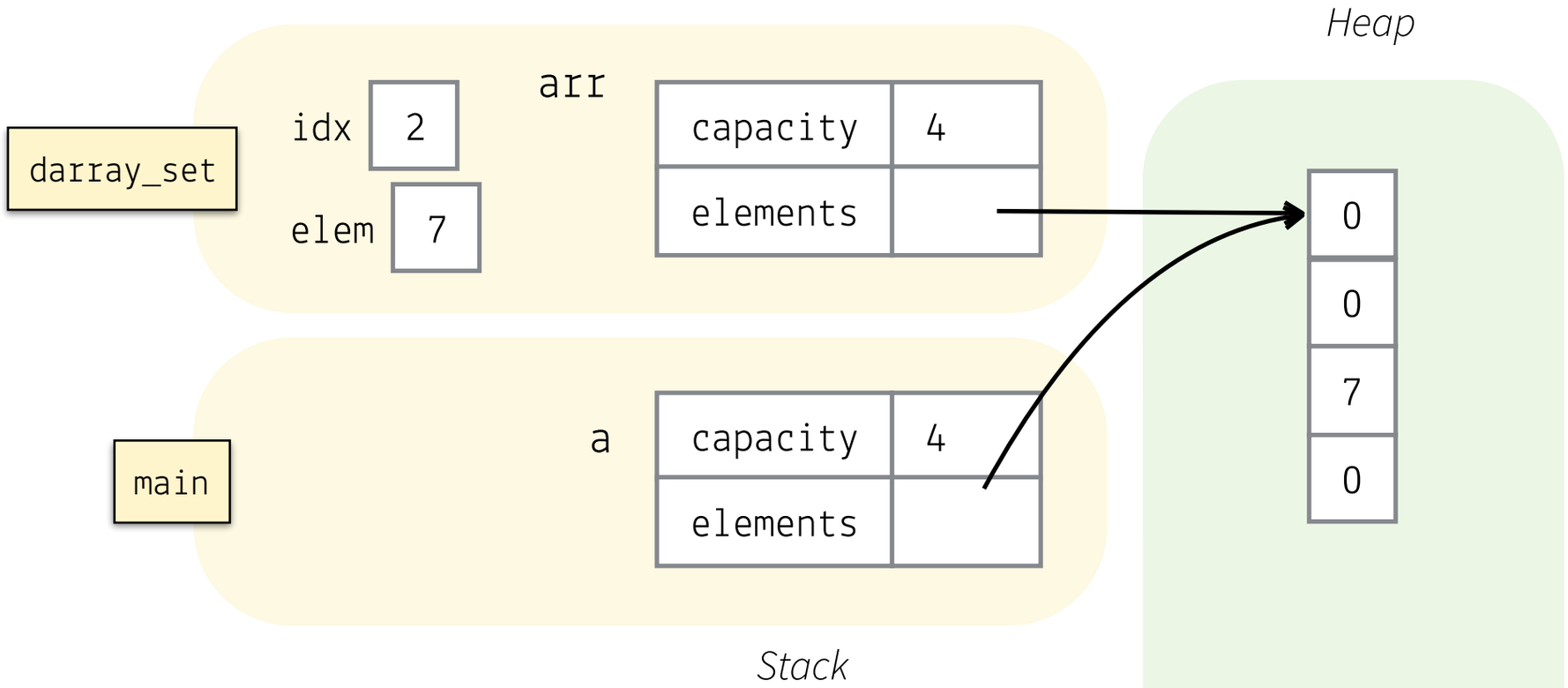
Ändra på minnesstorlek!

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



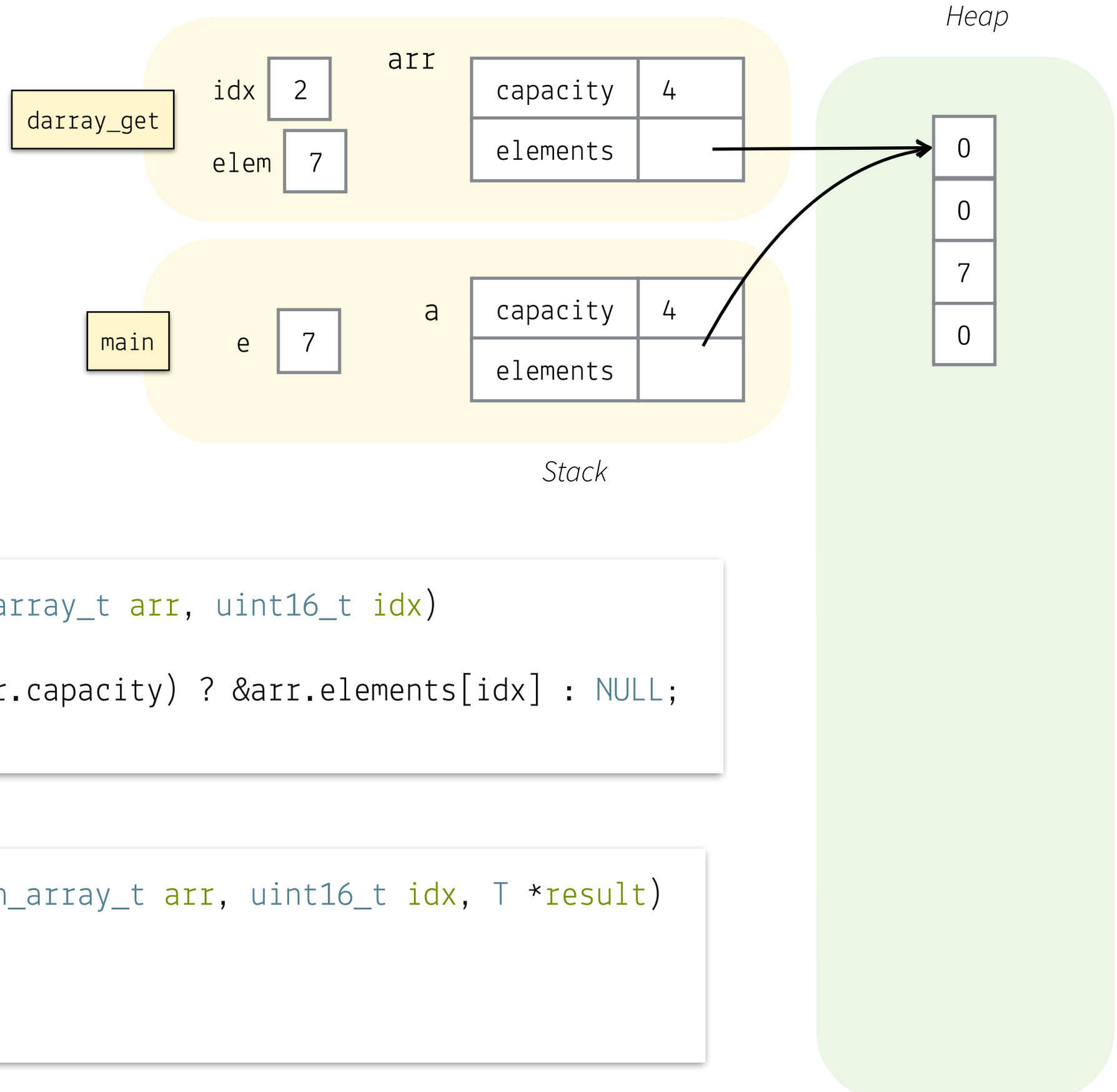

```
dyn_array_t darray_create(uint16_t capacity)
{
  // OBS! Borde göra felkontroll!
  return (dyn_array_t) {
    .capacity = capacity,
      .elements = calloc(capacity, sizeof(T)) };
}

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;
}
```

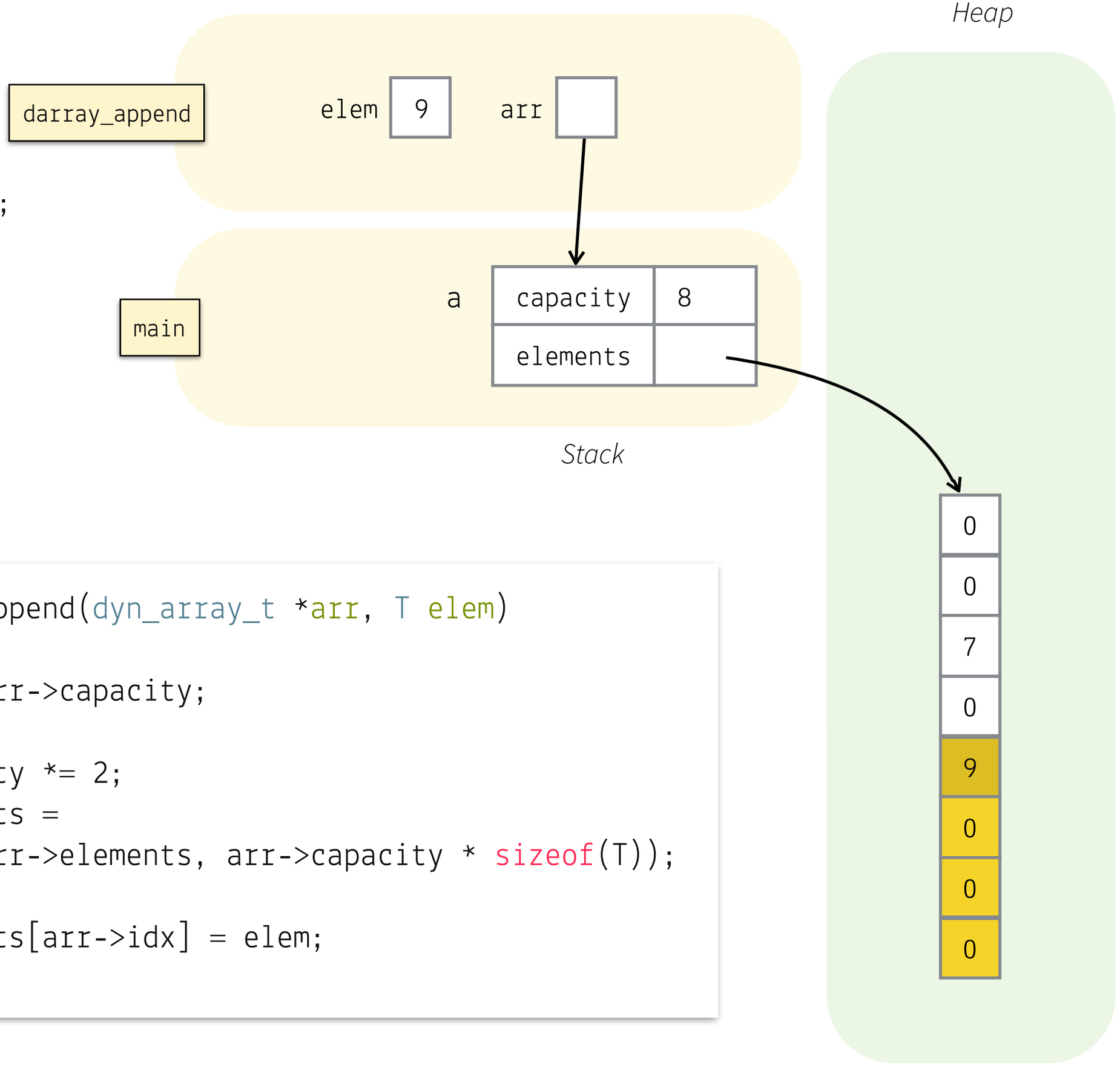
```
e = *darray_get(a, 2);
```




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

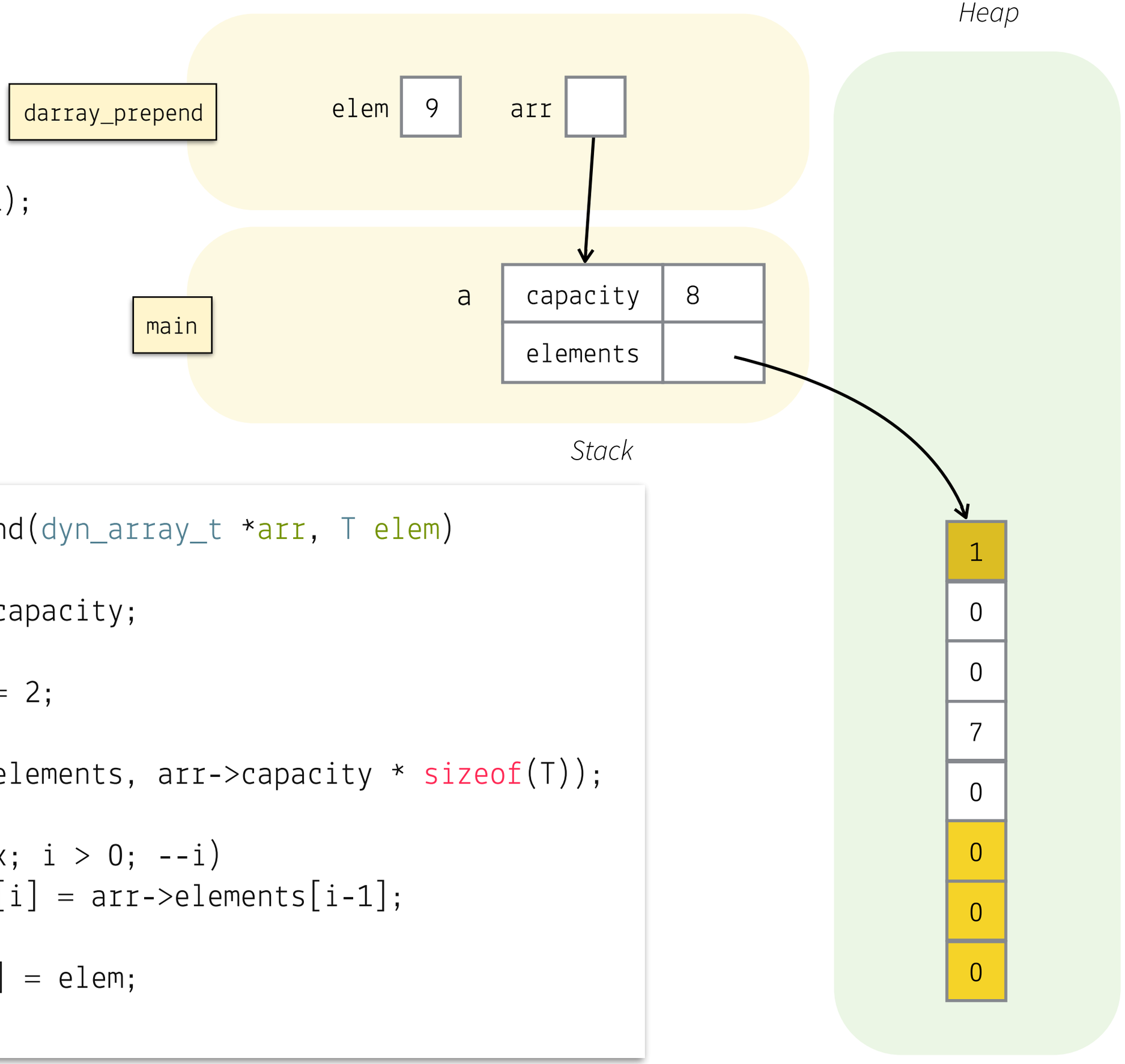
```
bool darray_get(dyn_array_t arr, uint16_t idx, T *result)
{
  ... // övning!
}
```

```
darray_append(&a, 9);
```




```
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);
```




```
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

Övningsuppgift hemma

- Varför används pekarsemantik ibland och värdesemantik ibland?

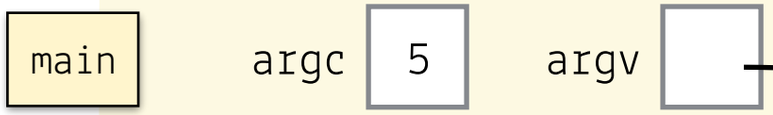
  Vad skulle hända om man bytte från pekarsemantik till värdesemantik eller tvärtom i t.ex. `darray_prepend`?

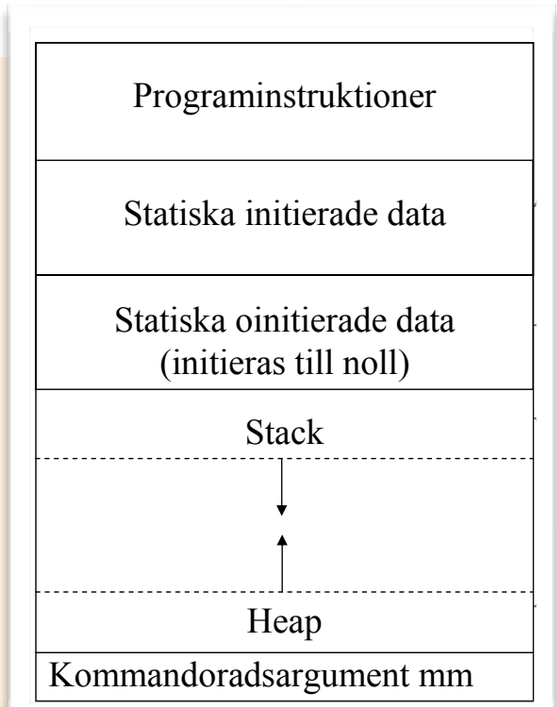
- Hur fungerar `malloc`, `free`, `calloc` och `realloc`?

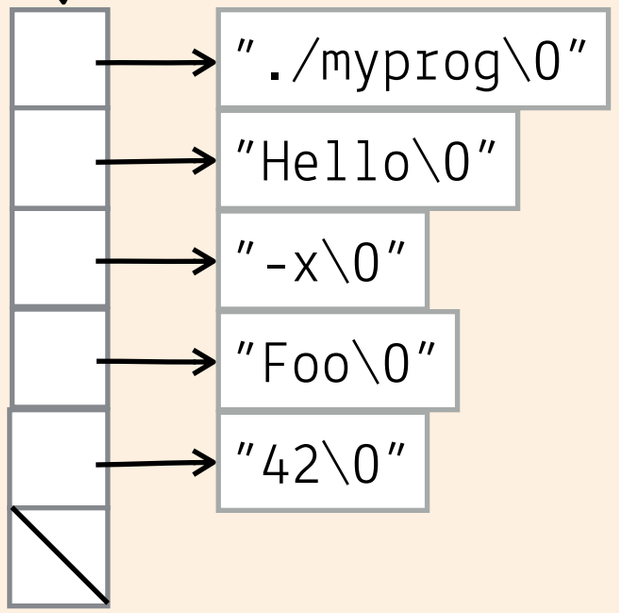
  Läs gärna man-sidorna (`$ man calloc`) så du har koll på man till kodprovet!

- 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 *`

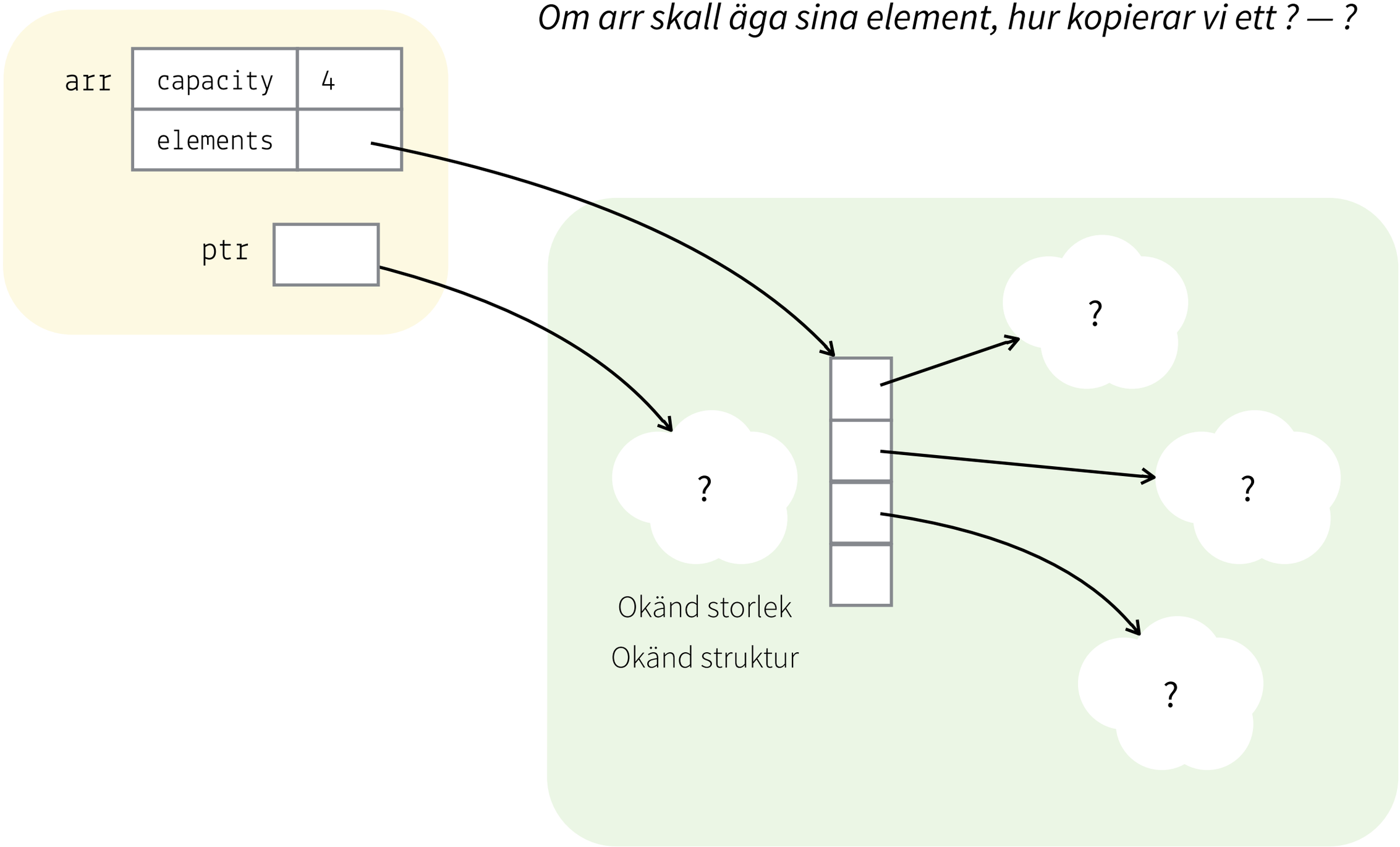
- Använda **`typedef void`** `*T;`

  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);
}
```

Vi skall se en lösning på detta på föreläsning 10!