

# **F3**

Stack & heap, manuell  
minneshantering, pekare

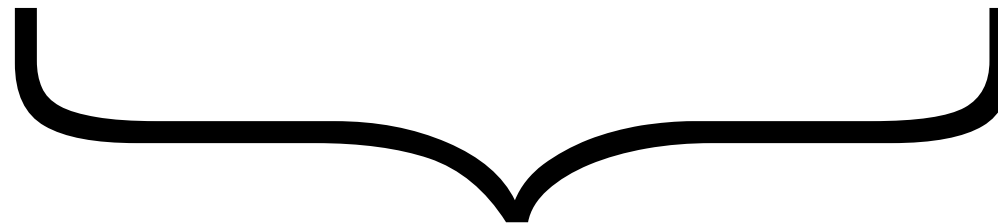
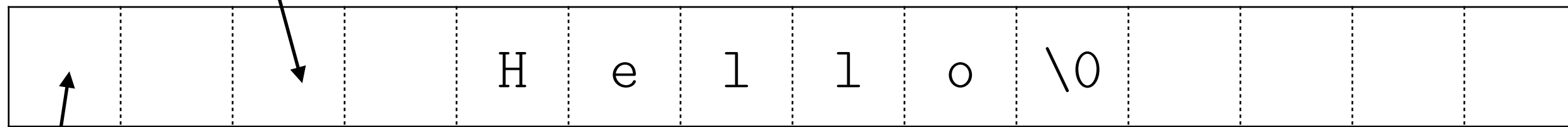
Tobias Wrigstad  
IOOPM 2014

- Har du tittat på >5 skärmföreläsningar?
- Har du tittat på >10 skärmföreläsningar?
- Har du tittat på skärmföreläsningen som introducerar ämnet för denna föreläsning?
- Kan du programmera C sedan tidigare?
- Är du bekant med begreppen stack och heap sedan tidigare?

Stacken

# C:s minnesmodell

Varje ruta är en byte



6 bytes

Varje byte i minnet har  
en **adress** — dess  
avstånd från "starten"

# Ett enkelt C-program & dess minnesanvändning

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

uint32_t fakultet(uint8_t f)
{
    if (f > 1)
    {
        return f * fakultet(f-1);
    }
    else
    {
        return 1;
    }
}

int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = fakultet(n);

    printf("%d! = %d\n", n, resultat);

    return 0;
}
```

4 bytes

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

uint32_t fakultet(uint8_t f)
{
    if (f > 1)
    {
        return f * fakultet(f-1);
    }
    else
    {
        return 1;
    }
}

int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = fakultet(n);

    printf("%d! = %d\n", n, resultat);

    return 0;
}
```

1 byte

n bytes

1 byte

? bytes

4 bytes

+binären, etc.

```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

uint32_t fakultet(uint8_t f)
{
    if (f > 1)
    {
        return f * fakultet(f-1);
    }
    else
    {
        return 1;
    }
}

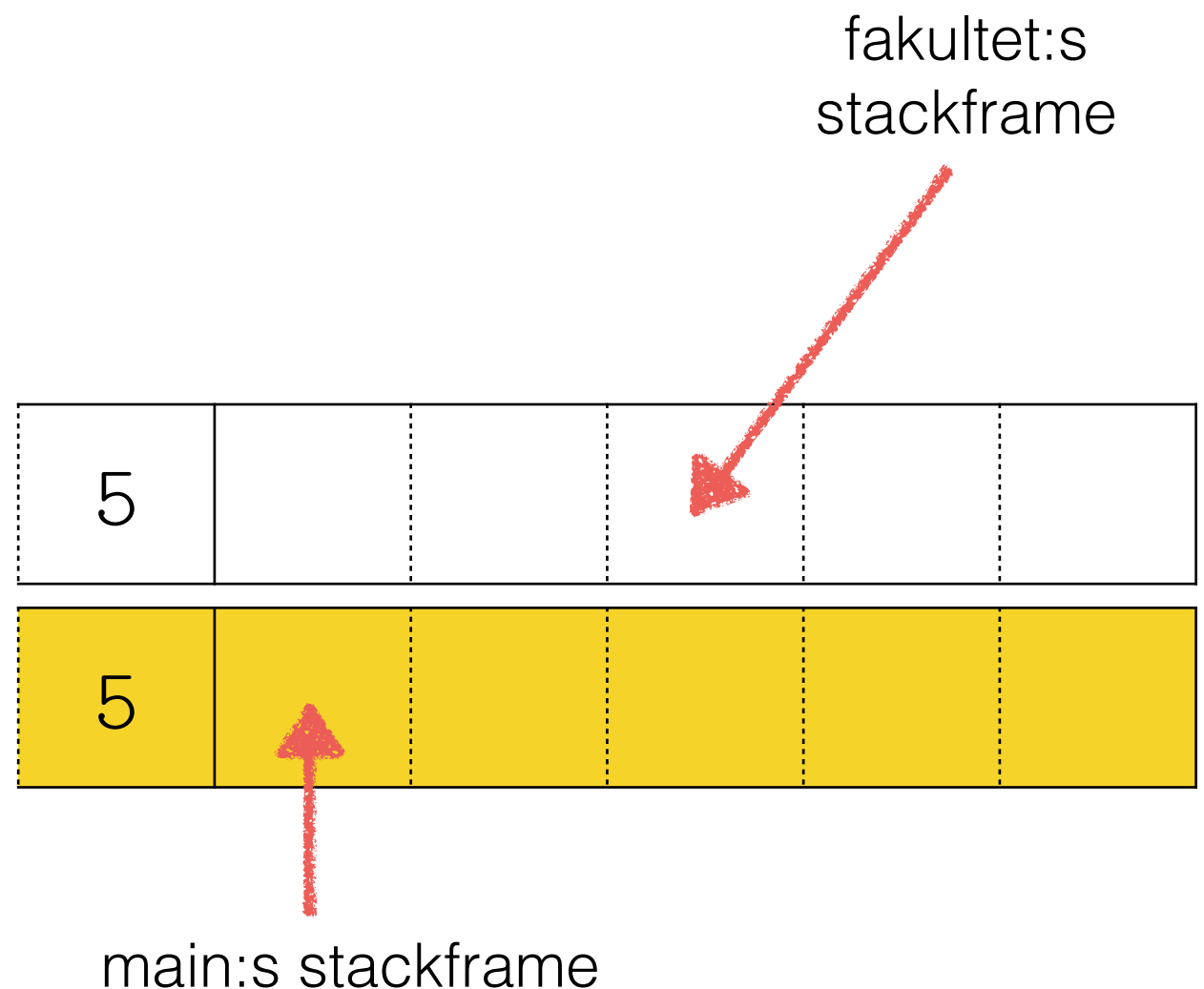
int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = fakultet(n);

    printf("%d! = %d\n", n, resultat);

    return 0;
}

```

f      <return value>



```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

uint32_t fakultet(uint8_t f)
{
    if (f > 1)
    {
        return f * fakultet(f-1);
    }
    else
    {
        return 1;
    }
}

int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = fakultet(n);

    printf("%d! = %d\n", n, resultat);

    return 0;
}

```

f                      <return value>

4					
5					
5					



```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

uint32_t fakultet(uint8_t f)
{
    if (f > 1)
    {
        return f * fakultet(f-1);
    }
    else
    {
        return 1;
    }
}

int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = fakultet(n);

    printf("%d! = %d\n", n, resultat);

    return 0;
}

```

f                      <return value>

3					
4					
5					
5					

```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

uint32_t fakultet(uint8_t f)
{
    if (f > 1)
    {
        return f * fakultet(f-1);
    }
    else
    {
        return 1;
    }
}

int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = fakultet(n);

    printf("%d! = %d\n", n, resultat);

    return 0;
}

```

f                      <return value>

2					
3					
4					
5					
5					

```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

uint32_t fakultet(uint8_t f)
{
    if (f > 1)
    {
        return f * fakultet(f-1);
    }
    else
    {
        return 1;
    }
}

int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = fakultet(n);

    printf("%d! = %d\n", n, resultat);

    return 0;
}

```

f	<return value>				
1					1
2					
3					
4					
5					
5					

```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

uint32_t fakultet(uint8_t f)
{
    if (f > 1)
    {
        return f * fakultet(f-1);
    }
    else
    {
        return 1;
    }
}

int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = fakultet(n);

    printf("%d! = %d\n", n, resultat);

    return 0;
}

```

f	<return value>				
1					1
2					2
3					
4					
5					
5					

```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

uint32_t fakultet(uint8_t f)
{
    if (f > 1)
    {
        return f * fakultet(f-1);
    }
    else
    {
        return 1;
    }
}

int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = fakultet(n);

    printf("%d! = %d\n", n, resultat);

    return 0;
}

```

f	<return value>				
1					1
2					2
3					6
4					
5					
5					

```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

uint32_t fakultet(uint8_t f)
{
    if (f > 1)
    {
        return f * fakultet(f-1);
    }
    else
    {
        return 1;
    }
}

int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = fakultet(n);

    printf("%d! = %d\n", n, resultat);

    return 0;
}

```

f	<return value>				
1					1
2					2
3					6
4					24
5					
5					

```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

uint32_t fakultet(uint8_t f)
{
    if (f > 1)
    {
        return f * fakultet(f-1);
    }
    else
    {
        return 1;
    }
}

int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = fakultet(n);

    printf("%d! = %d\n", n, resultat);

    return 0;
}

```

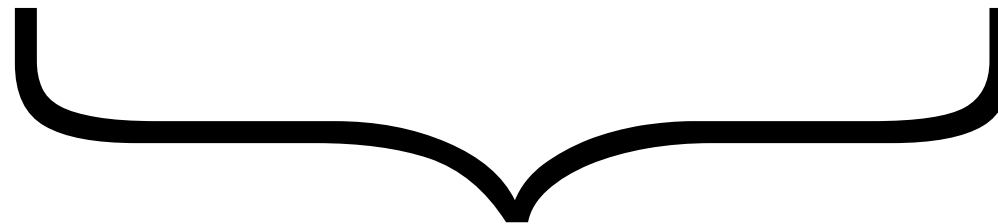
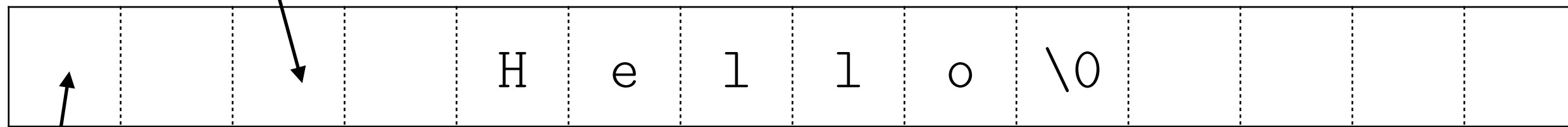
f	<return value>				
1					1
2					2
3					6
4					24
5					120
5					

”Pekare”



# C:s minnesmodell

Varje ruta är en byte



6 bytes

Varje byte i minnet har  
en **adress** — dess  
avstånd från "starten"

# Pekare

- Variabler som innehåller adresser till platser i minnet
- Två operationer:
  - Peka om variabeln
  - Avreferera pekaren för att läsa/skriva minnesplatsen

# Pekare

```
int x = 42;    // x innehåller ett heltal
```

# Pekare

```
int x = 42;    // x innehåller ett heltal
```

```
int *p;
```

# Pekare

```
int x = 42;    // x innehåller ett heltal
```

```
                // p innehåller en adress till en  
int *p;        // plats i minnet där det finns ett  
                // heltal
```

```
p = &x;
```

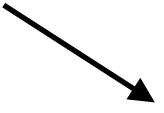
# Pekare

```
int x = 42;    // x innehåller ett heltal
```

```
int *p;
```

```
// p innehåller en adress till en  
// plats i minnet där det finns ett  
// heltal
```

adresstagnings-  
operatorn

 `p = &x;`

# Pekare

```
int x = 42;    // x innehåller ett heltal
```

```
int *p;
```

```
// p innehåller en adress till en  
// plats i minnet där det finns ett  
// heltal
```

adresstagnings-  
operatorn

```
p = &x;
```

```
// uppdatera pekaren p
```

```
*p = x;
```

# Pekare

```
int x = 42;    // x innehåller ett heltal
```

```
int *p;
```

```
// p innehåller en adress till en  
// plats i minnet där det finns ett  
// heltal
```

adresstagnings-  
operatorn

```
p = &x;
```

```
// uppdatera pekaren p
```

avrefererings-  
operatorn

```
*p = x;
```



# Pekare

```
int x = 42;    // x innehåller ett heltal
```

```
int *p;
```

```
// p innehåller en adress till en  
// plats i minnet där det finns ett  
// heltal
```

adresstagnings-  
operatorn

```
p = &x;
```

```
// uppdatera pekaren p
```

```
*p = x;
```

avrefererings-  
operatorn

```
// uppdatera minnesplatsen som p  
// pekar på
```

# ”Nullpekare” (1/2)

- ”I call it my billion-dollar mistake. It was the invention of the null reference in 1965” — Tony Hoare
- Syntax: `NULL`
- Semantik: variabeln pekar inte på någonting

```
int64_t *x = NULL;
```

# ”Nullpekare” (2/2)

```
int *x = NULL;
```

```
int y = *x;
```

# Pekare: recap

`&var` — ta adressen till innehållet i `var`

`*var` — följ en pekare till en plats i minnet som håller data

`NULL` — en pekare som inte pekar på någonting

Återbesök av  
stackexemplet med  
pekare

```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

void fakultet(uint8_t f, uint32_t *r)
{
    if (f > 1)
    {
        *r *= f;
        fakultet(f-1, r);
    }
}

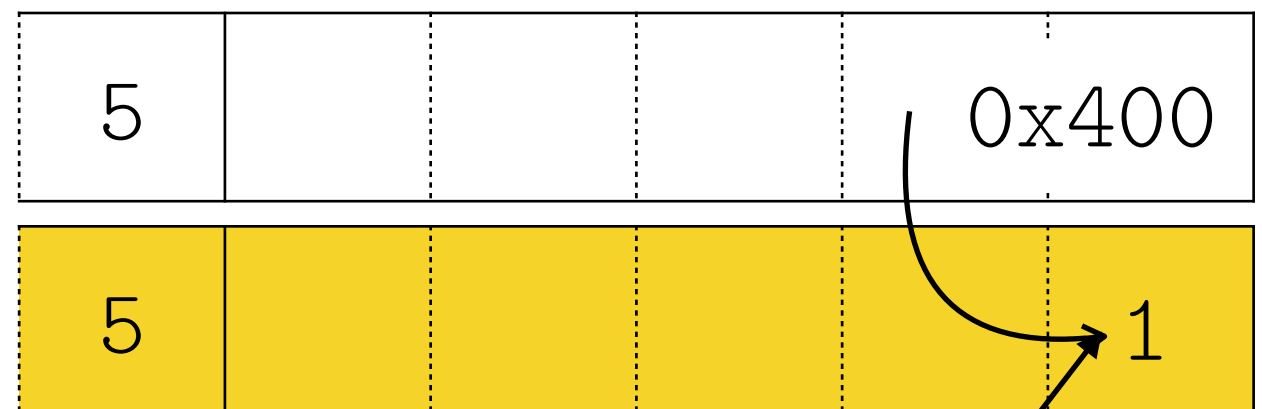
int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = 1;
    fakultet(n, &resultat);

    printf("%d! = %d\n", n, resultat);

    return 0;
}

```

f                      <return value>



0x400

```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

void fakultet(uint8_t f, uint32_t *r)
{
    if (f > 1)
    {
        *r *= f;
        fakultet(f-1, r);
    }
}

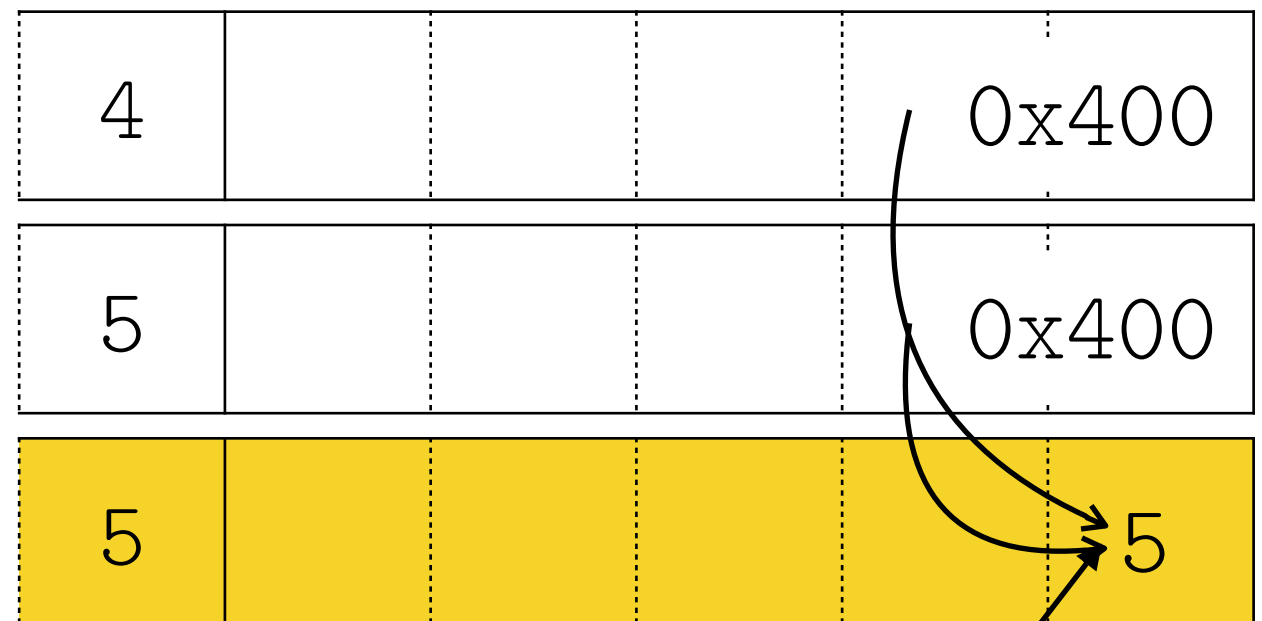
int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = 1;
    fakultet(n, &resultat);

    printf("%d! = %d\n", n, resultat);

    return 0;
}

```

f                      <return value>



0x400

```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

void fakultet(uint8_t f, uint32_t *r)
{
    if (f > 1)
    {
        *r *= f;
        fakultet(f-1, r);
    }
}

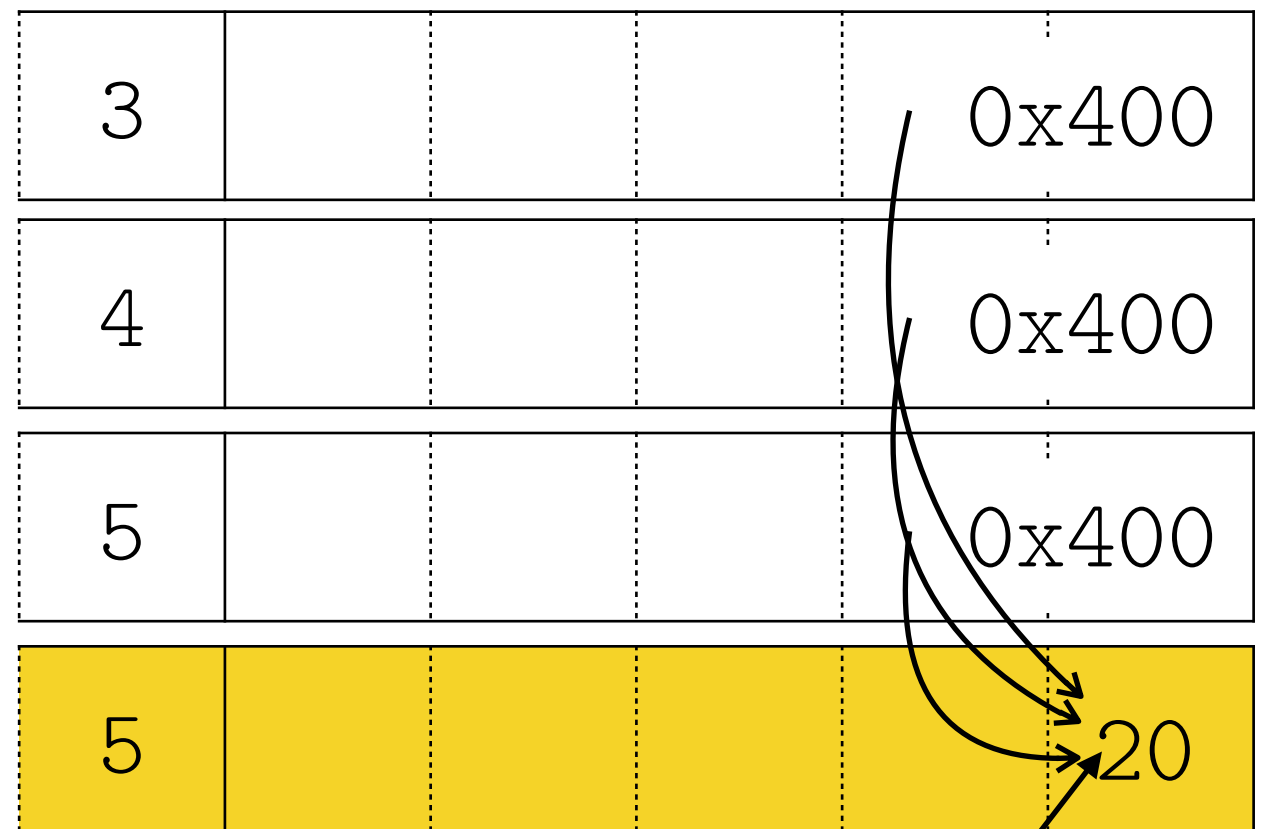
int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = 1;
    fakultet(n, &resultat);

    printf("%d! = %d\n", n, resultat);

    return 0;
}

```

f                      <return value>



0x400



```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

void fakultet(uint8_t f, uint32_t *r)
{
    if (f > 1)
    {
        *r *= f;
        fakultet(f-1, r);
    }
}

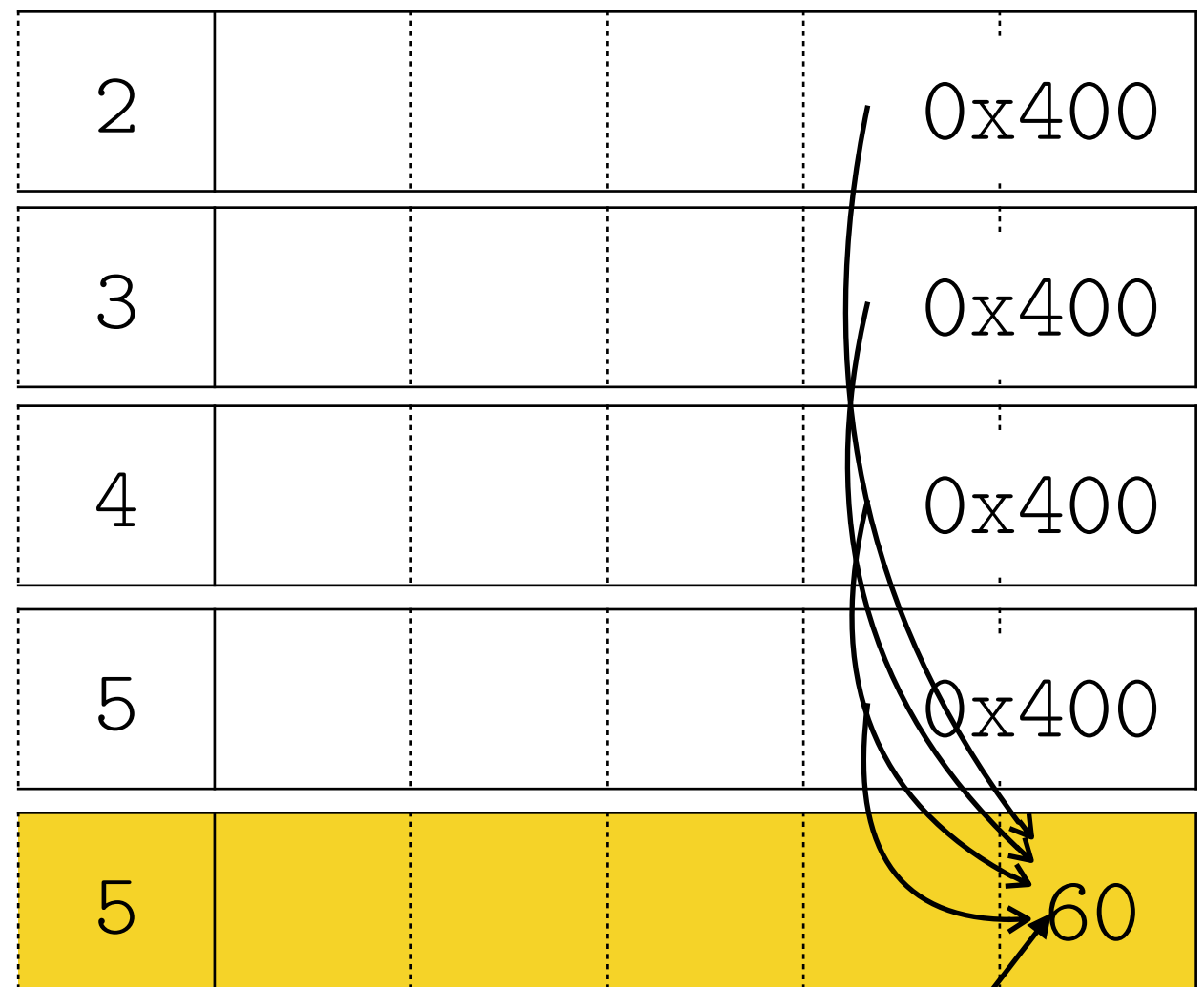
int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = 1;
    fakultet(n, &resultat);

    printf("%d! = %d\n", n, resultat);

    return 0;
}

```

f                      <return value>



0x400

```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

void fakultet(uint8_t f, uint32_t *r)
{
    if (f > 1)
    {
        *r *= f;
        fakultet(f-1, r);
    }
}

int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = 1;
    fakultet(n, &resultat);

    printf("%d! = %d\n", n, resultat);

    return 0;
}

```

f	<return value>			
1				0x400
2				0x400
3				0x400
4				0x400
5				0x400
5				120

0x400

```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

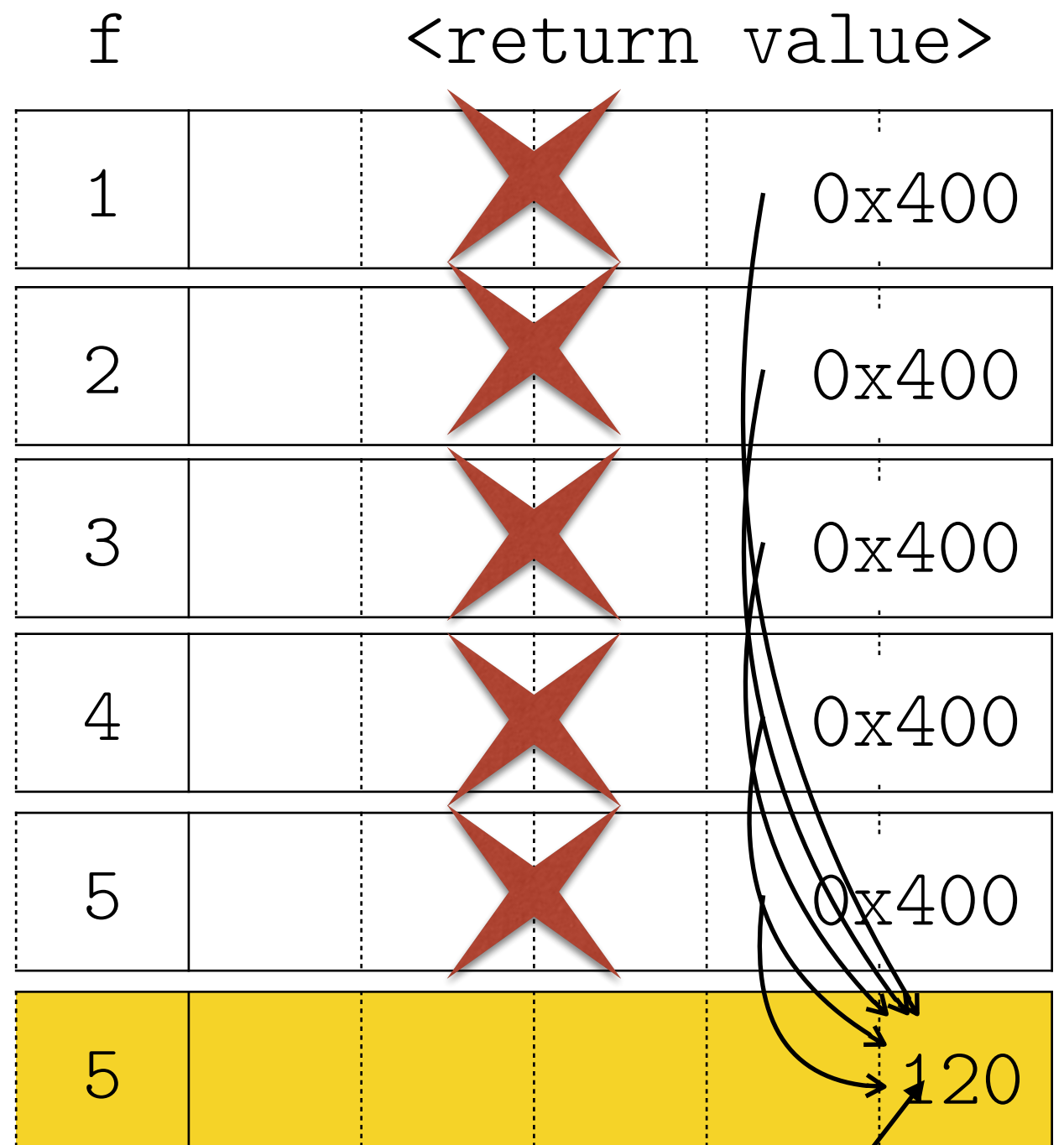
void fakultet(uint8_t f, uint32_t *r)
{
    if (f > 1)
    {
        *r *= f;
        fakultet(f-1, r);
    }
}

int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = 1;
    fakultet(n, &resultat);

    printf("%d! = %d\n", n, resultat);

    return 0;
}

```



0x400

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

void fakultet(uint8_t f, uint32_t *r)
{
    for (int i = 0; i < f; ++i)
    {
        *r *= f-i;
    }
}

int main(int argc, char *argv[])
{
    uint8_t n = atoi(argv[1]);
    uint32_t resultat = 1;
    fakultet(n, &resultat);

    printf("%d! = %d\n", n, resultat);

    return 0;
}
```

Kör i konstant minne  
(varför?)

# Stacken: recap

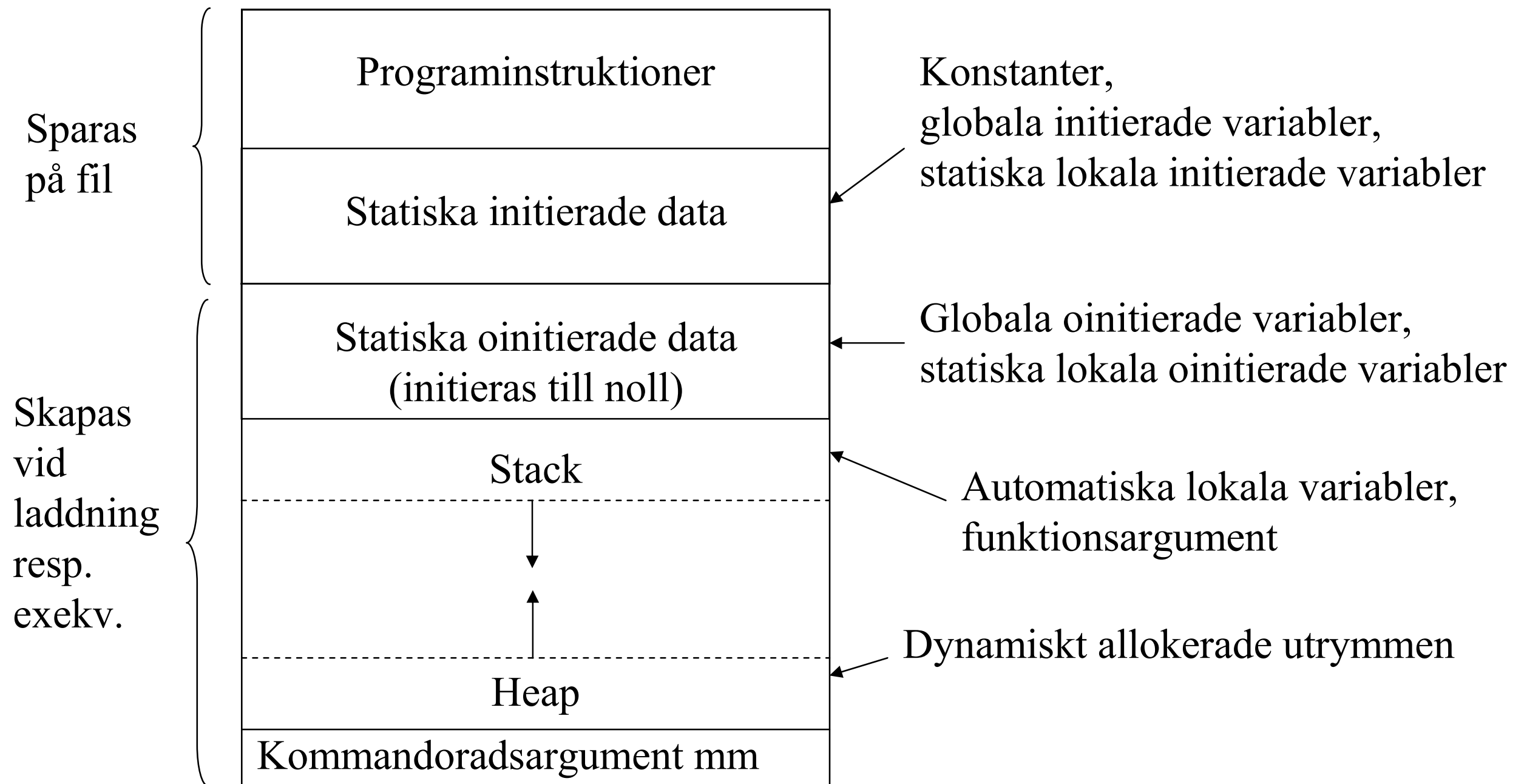
- Enda minnet som C:s "exekveringsmiljö" hanterar automatiskt
  - Håller "kortlivat data" (knutet till funktionens livslängd)
  - Varje funktionsanrop ger upphov till en ny "stack frame" (not. parameteröverföring sker i regel mha register)
  - Allokeras och avallokeras automatiskt
- Extremt effektiv allokeringssmetod ("bump pointer")
- Stacken är en del av minnet — alla variabler på stacken har en adress och kan pekars ut av pekare

Heapen

# Heapen

- Till för lagring av "långlivat" data
- Hanteras manuellt
  - För varje data som skall lagras på heapeen måste man explicit be om ett motsvarande utrymme mha `malloc` (även `realloc` och `calloc`)
  - När man är färdig med data måste man explicit returnera det, annars läcker programmet minne mha funktionen `free`
- Heapeen bara tillgänglig via pekare

# Var lagras data?

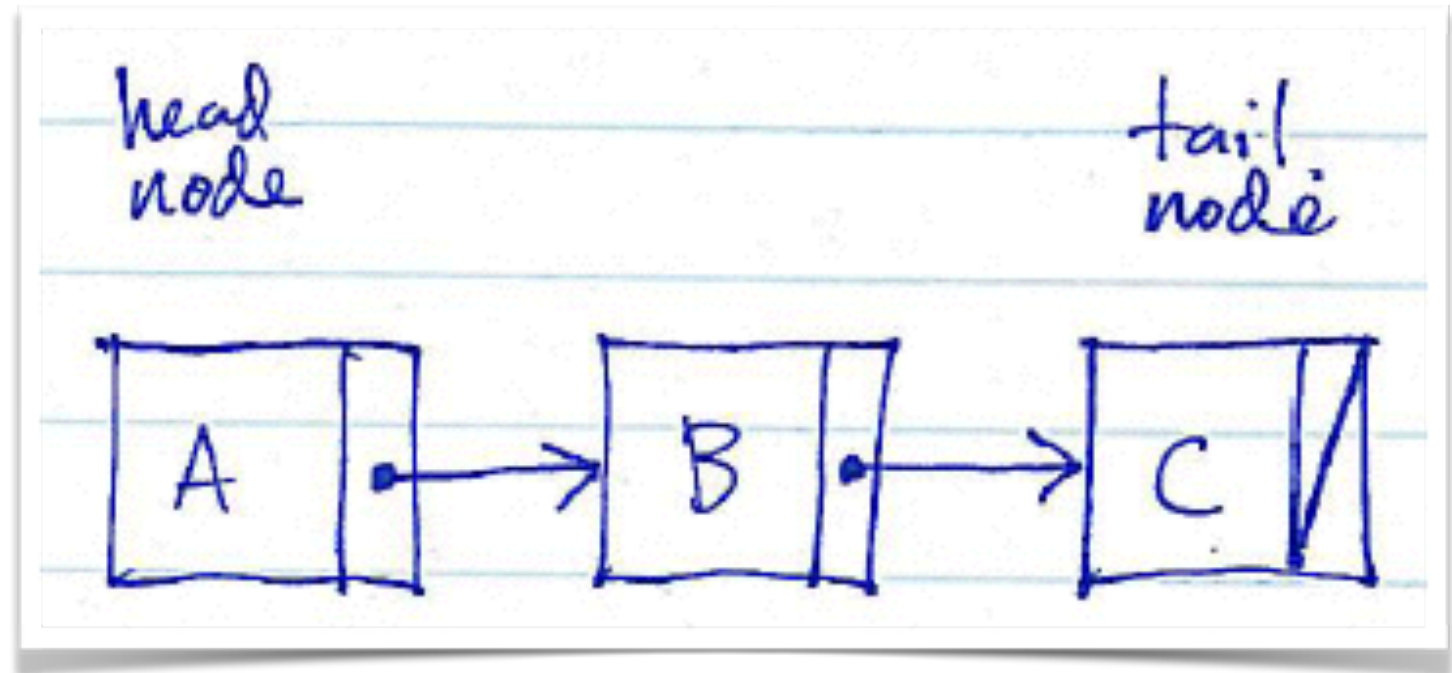




# Dynamiska strukturer

- Heapen används för att lagra stora data och data vars storlek växer dynamiskt, t.ex.
  - Binära träd, listor, köer, databaser, filbuffrar, etc.
- Vi använder malloc för att "reservera en del av heapen" — denna kan vara "var som helst"
- Pekare **nödvändiga** för att *länka samman* datastrukturer som byggs upp av **fler än ett** anrop till malloc!

## En "länkad" lista



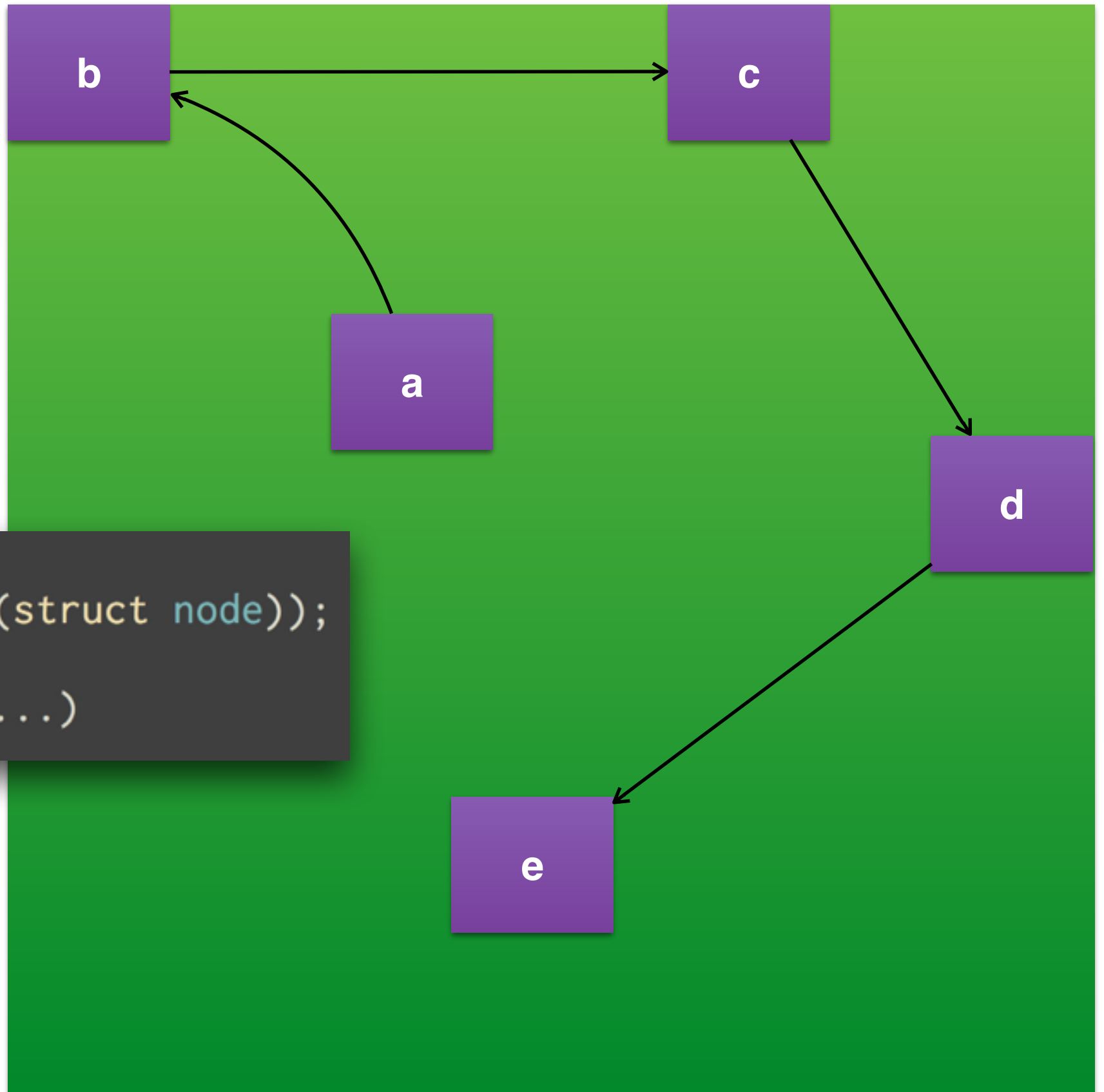
```
struct node
{
    int64_t number;
    struct node *next;
};
```

```
node *n;
n->next = malloc(sizeof(struct node));
n->number = 42;
n->next->next = malloc(...)
```

En "länkad" lista

```
struct node
{
    int64_t number;
    struct node *next;
};
```

```
node *n;
n->next = malloc(sizeof(struct node));
n->number = 42;
n->next->next = malloc(...)
```



# Avreferera pekare

`(*node).next = node->next`



Följ pekaren



Läs next-posten  
i strukten

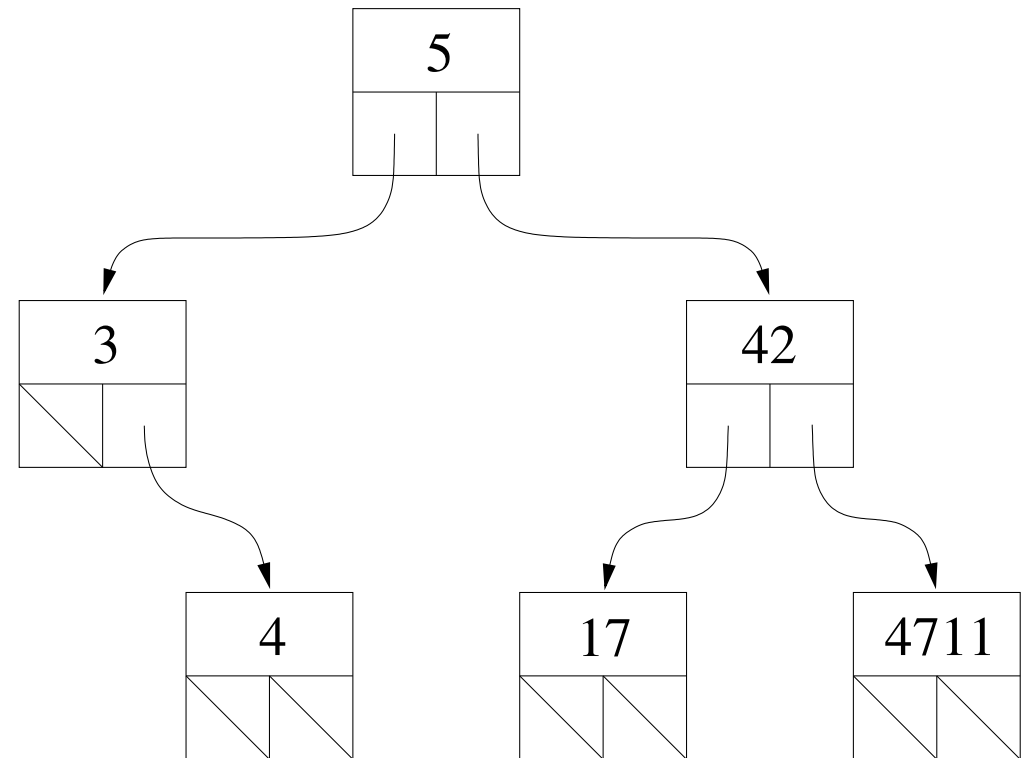


Båda i samma  
operator

# Pekare och länkade strukturer

```
typedef struct node {  
    int key;  
    struct node *left;  
    struct node *right;  
} node, *Link;
```

```
int cons(int key) {  
    Link result = (Link) malloc(sizeof(node));  
    result->key = key;  
    result->left = NULL;  
    result->right = NULL;  
    return result;  
}
```



## Att skapa noder i ett träd

```
typedef struct bst_node
{
    node_t *left;
    node_t *right;
    some_data_t *data;
} node_t;
```

```
node_t node_new(some_data_t *d)
{
    node_t *new_node = malloc(sizeof(node_t));
    if (new_node)
    {
        new_node->left = NULL;
        new_node->right = NULL;
        new_node->data = d; // aliasing!
    }
    return new_node;
}
```

```
void node_rm(node_t *a)
{
    if (a->left != NULL) node_rm(a->left);
    if (a->right != NULL) node_rm(a->right);
    free(a);
}
```

```
void node_rm_better(node_t *a)
{
    if (a->left != NULL) node_rm(a->left);
    if (a->right != NULL) node_rm(a->right);
    free(a->data);
    free(a);
}
```

**Ta bort  
noder  
ur ett träd**

# Minneshantering

- Allokera minne explicit med ngn rutin (t.ex. malloc)
- Frigör minne explicit med ngn rutin (t.ex. free)
- Vem för bok över vilket minne som är ledigt resp. använt?
- Hur hittar vi ett lämpligt ledigt utrymme?
- Vad betyder lämpligt?



**256 k**

```
a = malloc(A);
```

```
b = malloc(B);
```

```
c = malloc(C);
```

```
d = malloc(D);
```

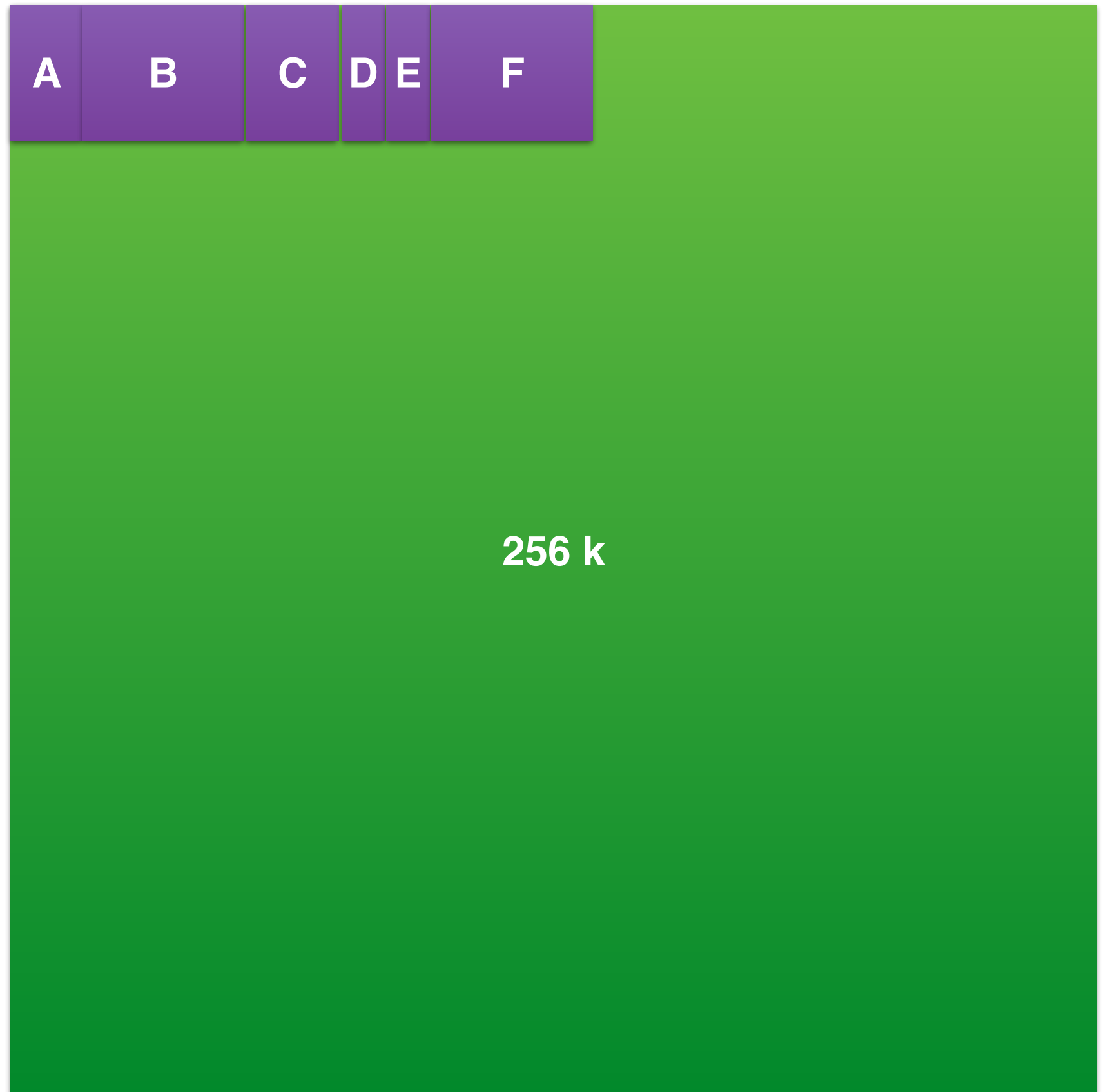
```
e = malloc(E);
```

```
f = malloc(F);
```

```
free(a);
```

```
free(c);
```

```
g = malloc(B);
```



```
a = malloc(A);
```

```
b = malloc(B);
```

```
c = malloc(C);
```

```
d = malloc(D);
```

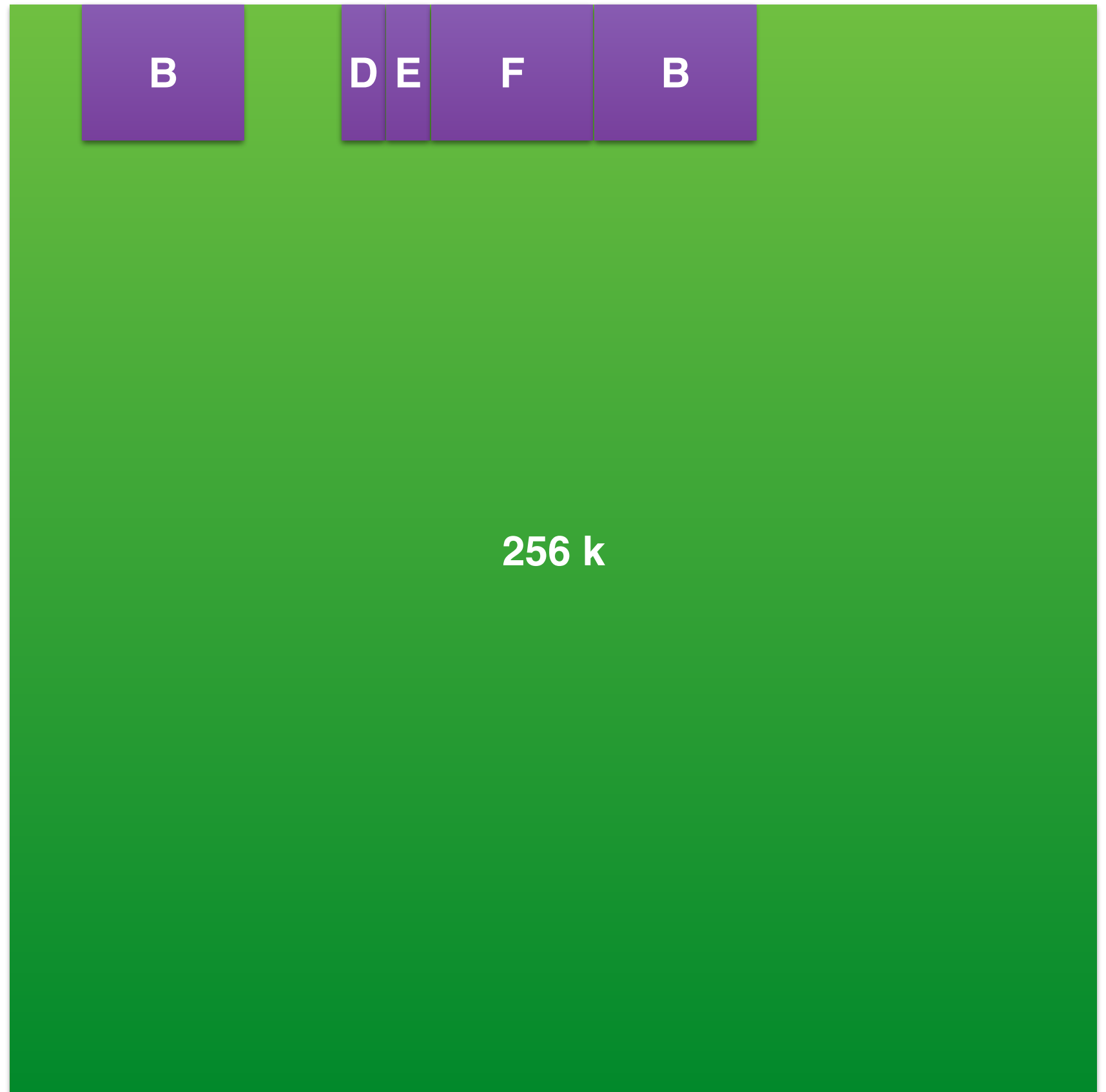
```
e = malloc(E);
```

```
f = malloc(F);
```

```
free(a);
```

```
free(c);
```

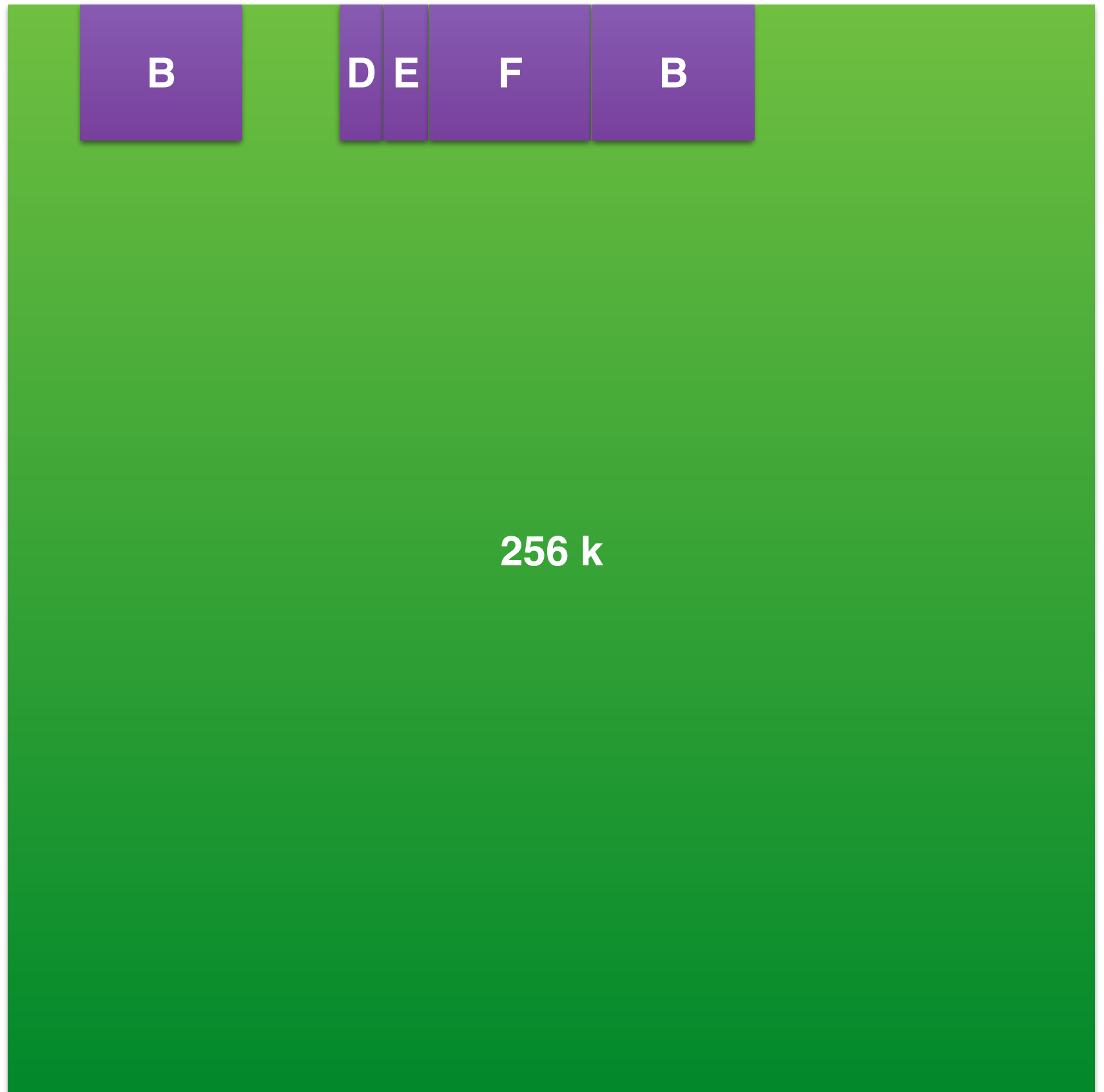
```
g = malloc(B);
```



## Fragmentering

Vi fick inte rum med B' i det lediga minnet från A och C eftersom de inte var *sammanhängande* (konsekutiva; contiguous)

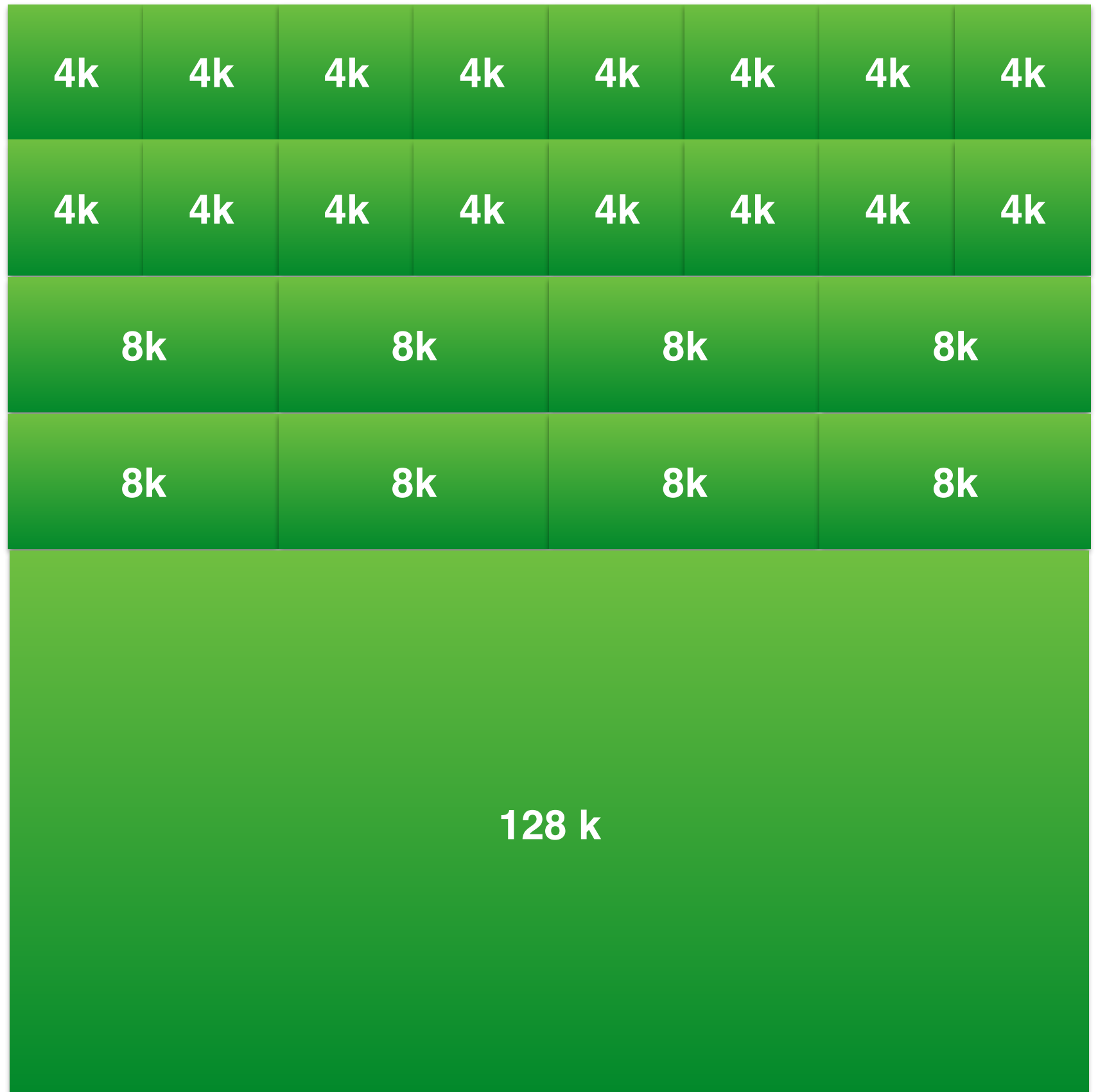
Kan leda till att minnet effektivt tar slut fast det finns gott om ledigt minne



## Bucket Allocation

Dela in minnet i många  
bitar av olika storlekar  
för snabbare allokering  
av objekt

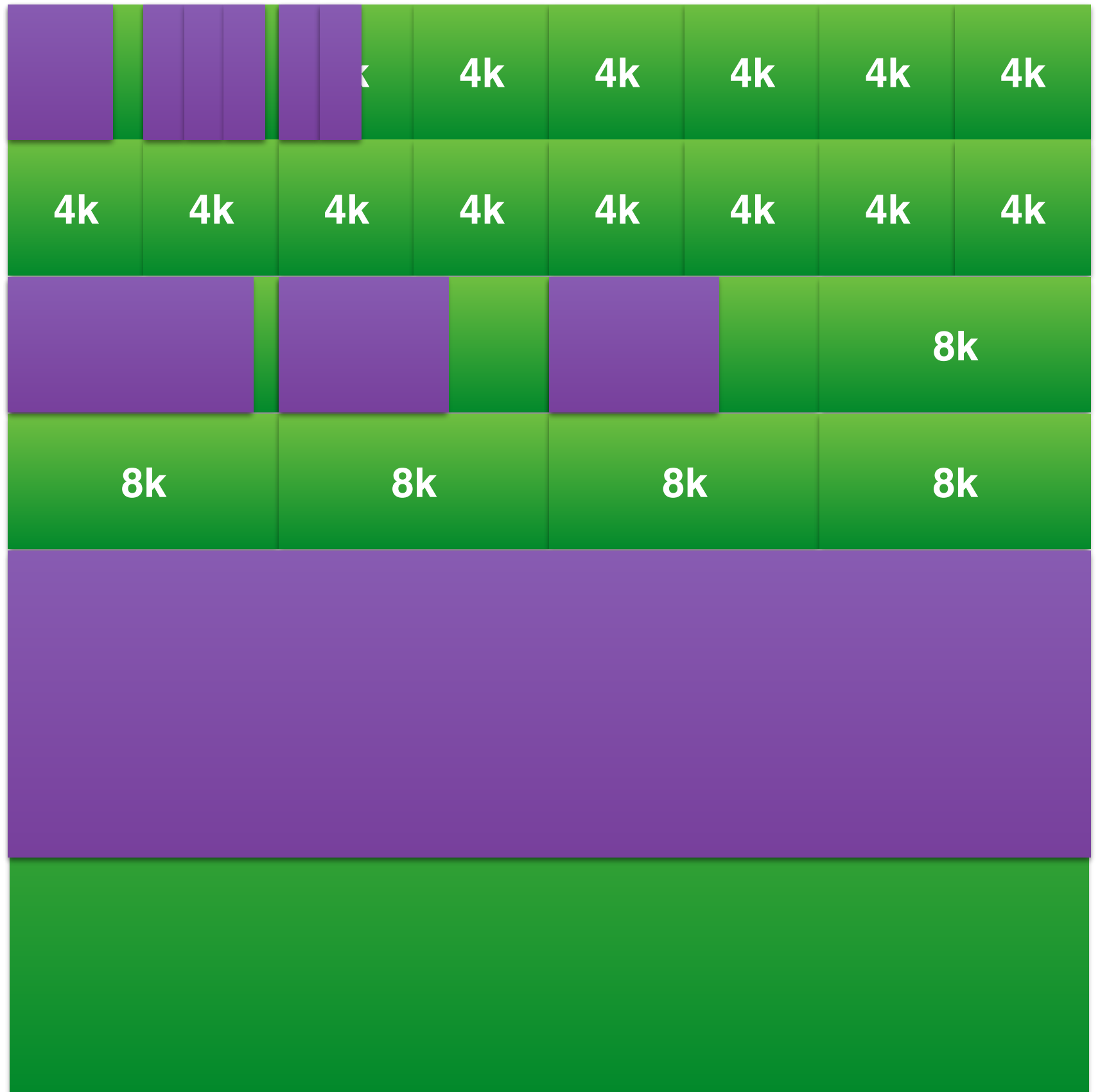
Vad får detta för effekt  
på fragmentering?



## Bucket Allocation

Dela in minnet i många  
bitar av olika storlekar  
för snabbare allokering  
av objekt

Vad får detta för effekt  
på fragmentering?



Hitta felet!

**Hitta  
felet!**

```
int8_t *add(int32_t count, int8_t *fst, int8_t *snd)
{
    int8_t *result = malloc(count * sizeof(int8_t));
    int8_t *cursor = result;
    while (count--) *cursor++ = *fst++ + *snd++;
    free(fst);
    free(snd);
    return result;
}
```



# Hitta felet!

2. Därför måste vi "spara undan" pekaren till startadressen

```
int8_t *add(int32_t count, int8_t *fst, int8_t *snd)
{
    int8_t *result = malloc(count * sizeof(int8_t));
    int8_t *cursor = result;
    while (count--) *cursor++ = *fst++ + *snd++;
    free(fst);
    free(snd);
    return result;
}
```

1. Vi flyttar pekaren i minnet

3. Vi glömde det mönstret för `fst` och `snd`!

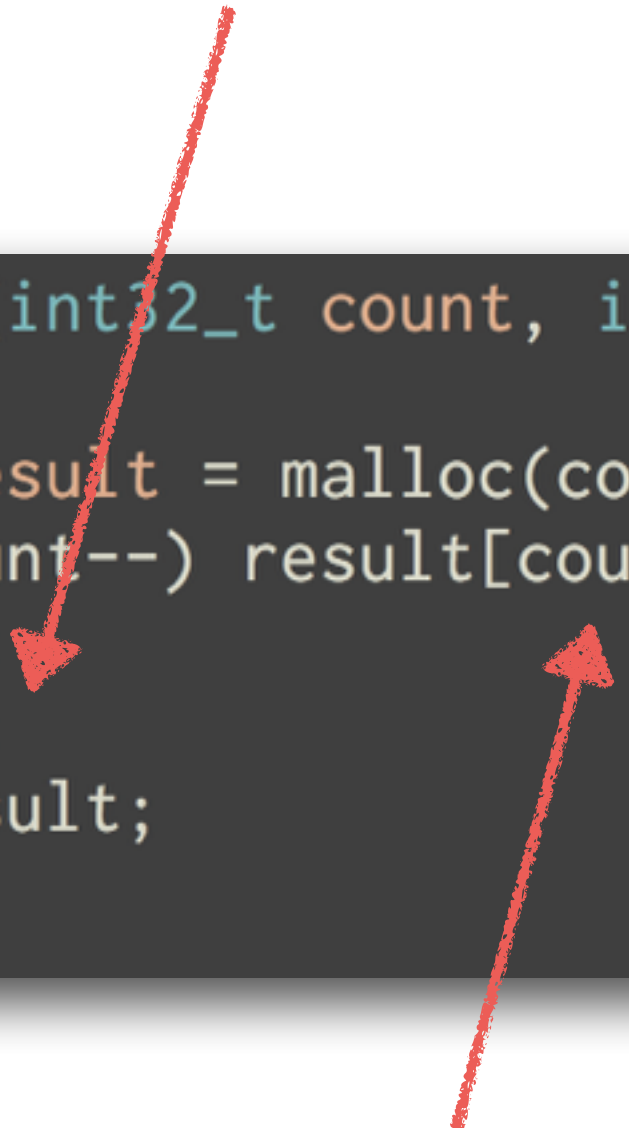
# Hitta felet!

```
int8_t *add(int32_t count, int8_t *fst, int8_t *snd)
{
    int8_t *result = malloc(count * sizeof(int8_t));
    while (count--) result[count] = fst[count] + snd[count];
    free(fst);
    free(snd);
    return result;
}
```

# Hitta felet!

2. Men vad händer om `fst == snd`?

```
int8_t *add(int32_t count, int8_t *fst, int8_t *snd)
{
    int8_t *result = malloc(count * sizeof(int8_t));
    while (count--) result[count] = fst[count] + snd[count];
    free(fst);
    free(snd);
    return result;
}
```



1. Kod med arrayindex är tydligare

# Aliasering

- "Två eller fler variabler avser samma objekt"
  - Förändring via en väg synlig genom en annan

```
*x = 1;  
*y = 0;  
printf("%d\n", *x);
```

Vad skrivs ut av detta program?

- Kraftfullt!
- Livsfarligt!
- Fundamentalt problem i programspråksfältet

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int8_t *add(int32_t count, int8_t *fst, int8_t *snd)
{
    int8_t *result = malloc(count * sizeof(int8_t));
    while (count--) result[count] = fst[count] + snd[count];
    free(fst);
    free(snd);
    return result;
}

int main(int argc, char *argv[])
{
    int8_t even[4] = { 2, 4, 6, 8 };
    int8_t odd[4] = { 1, 3, 5, 7 };

    int8_t *sum = add(4, odd, even);

    for (int i=0; i<4; ++i)
        printf("%d ", sum[i]);

    free(sum);
    return 0;
}
```

**Hitta  
felet!**

```

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int8_t *add(int32_t count, int8_t *fst, int8_t *snd)
{
    int8_t *result = malloc(count * sizeof(int8_t));
    while (count--) result[count] = fst[count] + snd[count];
    free(fst);
    free(snd);
    return result;
}

int main(int argc, char *argv[])
{
    int8_t even[4] = { 2, 4, 6, 8 };
    int8_t odd[4] = { 1, 3, 5, 7 };

    int8_t *sum = add(4, odd, even);

    for (int i=0; i<4; ++i)
        printf("%d ", sum[i]);

    free(sum);
    return 0;
}

```

## Hitta felet!

1. Samma kod som "sist"

2. even och odd är allokerade på stacken!

# Hitta felet!

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int8_t *add(int32_t count, int8_t *fst, int8_t *snd)
{
    int8_t *result = malloc(count * sizeof(int8_t));
    while (count--) result[count] = fst[count] + snd[count];
    free(fst);
    free(snd);
    return result;
}

int main(int argc, char *argv[])
{
    int8_t *numbers = malloc(4 * sizeof(int8_t));
    numbers[0] = 1; numbers[1] = 2; numbers[2] = 3; numbers[3] = 4;

    int8_t *sum = add(4, numbers, numbers);

    for (int i=0; i<4; ++i)
        printf("%d ", sum[i]);

    free(sum);
    return 0;
}
```



# Hitta felet!

1. add  
anropas  
"med  
samma  
array"

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int8_t *add(int32_t count, int8_t *fst, int8_t *snd)
{
    int8_t *result = malloc(count * sizeof(int8_t));
    while (count--) result[count] = fst[count] + snd[count];
    free(fst);
    free(snd);
    return result;
}

int main(int argc, char *argv[])
{
    int8_t *numbers = malloc(4 * sizeof(int8_t));
    numbers[0] = 1; numbers[1] = 2; numbers[2] = 3; numbers[3] = 4;

    int8_t *sum = add( numbers, numbers );

    for (int i=0; i<4; ++i)
        printf("%d ", sum[i]);

    free(sum);
    return 0;
}
```



## Manuell minneshantering är felbenäget

- *Vem* ansvarar för att avallokera?
- Hur vet jag om "*jag*" är ansvarig?
- Hur vet jag var minnet *går* att avallokera?

## Typiska fel

- Dubbel avallokering (double deallocation)
- Skjutna pekare (dangling pointers)
- Tappa bort pekaren till startadressen

# Minnet i C

- Datastrukturer av dynamisk storlek bor på heapen
  - I regel länkade strukturer (men även `realloc`)
- Inget skydd för överskrivning av data i ett program
- Måste se till att allokera nog med yta
- Måste själv anropa `free` i rätt tid
- Se valgrind för verktygsstöd för att hantera minne
- "malloc är inte magisk"