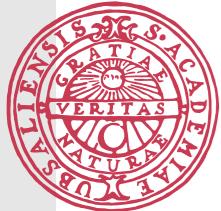


Föreläsning 3

Tobias Wrigstad

*Inför sista labben under
startup-veckorna!*



Databas?

```
struct databas
{
    struct vara varor[128];
    int antal_varor;
};

typedef struct databas db_t;

db_t db; // lokal variabel i main()
```

Hitta en vara i databasen?

- Vi måste ha tillgång till databasen
- Vi måste veta hur många varor som finns i den
- Vi måste ha en sökkriterium
- Vi måste veta hur man implementerar sökkriteriet

Hitta en vara i databasen?

- **Vi måste ha tillgång till databasen**

Skicka runt en pekare till databasen från main()

db->varor är en array av varor

- **Vi måste veta hur många varor som finns i den**

Ex. db->antal_varor

- **Vi måste ha en sökkriterium**

Strängjämförelse eller varans index i databasen

- **Vi måste veta hur man implementerar sökkriteriet**

strcmp eller jämförelse mellan intar == (etc.)

Hitta en vara i databasen?

```
bool finns_vara_i_databasen(char *varans_namn, db_t *db)
{
    int i = 0;

    while (i < db->antal_varor)
    {
        char *aktuellt_namn = db->varor[i].namn;
        // OBS! Använd strcmp istället, kolla hur!
        if (strcmp(varans_namn, aktuellt_namn) == 0)
        {
            return true;
        }
        i = i + 1;
    }

    return false;
}
```

Lägga till en vara i databasen

```
bool lagg_till_vara_i_databasen(struct vara v, db_t *db)
{
    if (db->antal_varor < 128)
    {
        db->varor[db->antal_varor] = v;
        db->antal_varor += 1;
        return true;
    }
    else
    {
        return false;
    }
}
```

Lägga till en vara i databasen (take 2)

```
bool lagg_till_vara_i_databasen(struct vara v, db_t *db)
{
    if (db->antal_varor < db->max)
    {
        db->varor[db->antal_varor++] = v;
        return true;
    }
    else
    {
        return false;
    }
}
```

Lägga till en vara i databasen (take 3)

```
bool lagg_till_vara_i_databasen(vara_t *v, db_t *db)
{
    if (db->antal_varor < db->max)
    {
        db->varor[db->antal_varor++] = v;
        return true;
    }
    else
    {
        return false;
    }
}
```

Varför inte?

värdesemantik!

```
bool lagg_till_vara_i_databasen(vara_t *v, db_t db)
{
    if (db.antal_varor < db.max)
    {
        db.varor[db.antal_varor++] = v;
        return true;
    }
    else
    {
        return false;
    }
}
```

PKD:n lärde ut ”function templates”

- **Hitta ett element i en array** är definitivt ett återkommande mönster

Kolla om elementet finns är nästan precis lika (men returnerar en bool istället)

Att anpassa funktionen på sidan innan är inte invecklat

Enda problemet: vad gör man om elementet inte finns?

Enkel lösning: returnera något element – kräv att man först kollar **om** det finns

Bättre lösning: returnera en pekare (NULL om elementet inte fanns)

Gruppmöten på fredag

- TBA

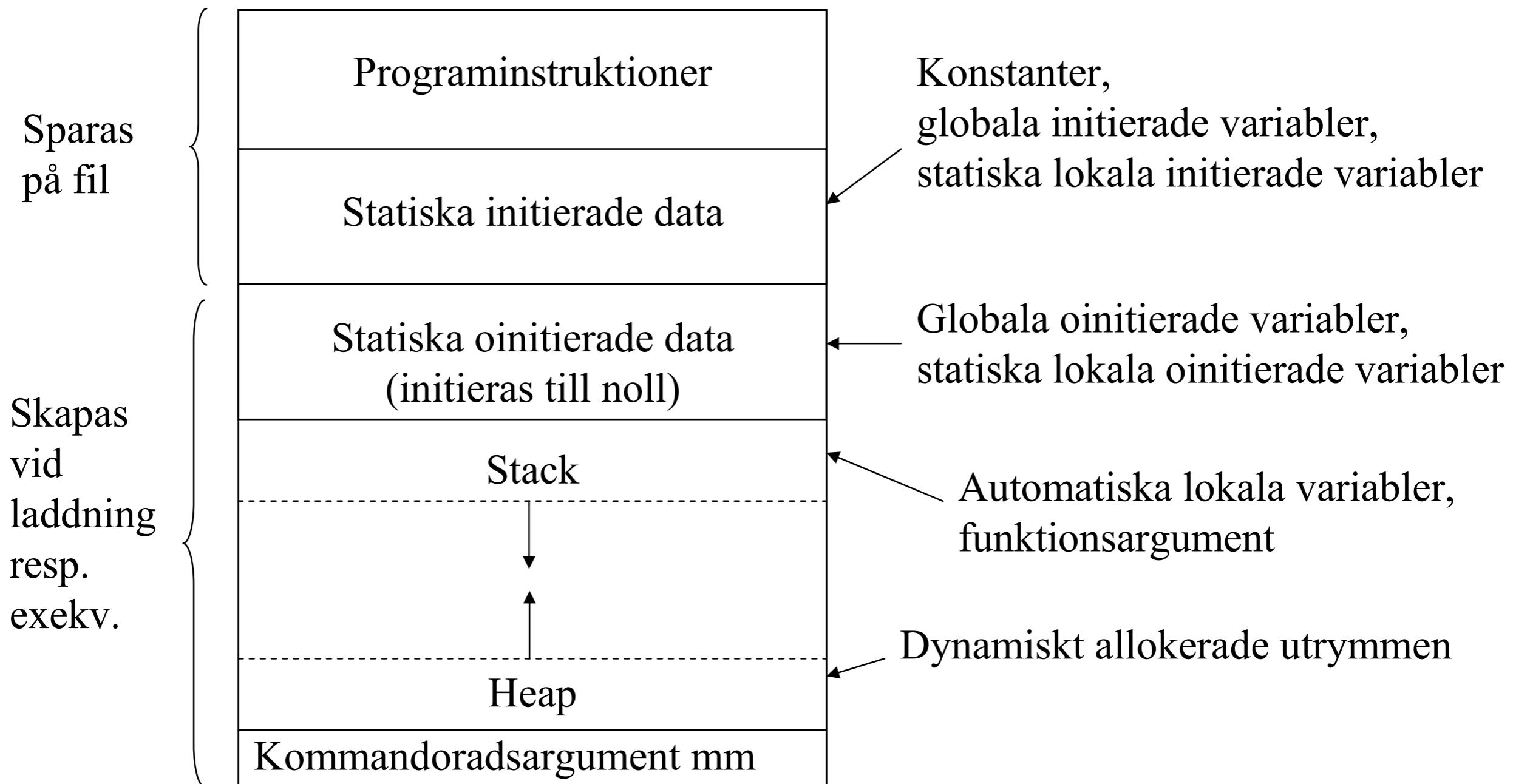
Föreläsning 3

Tobias Wrigstad

*Stack & heap, manuell
minneshantering, pekare*



Var lagras data?



Stacken

Läs Addendum om stack och heap



Ett enkelt stackexempel

Skrivet saso-det har lilla hattet är att driva hem några posturer kring skillnaderna mellan stack och heap, adressräkningsoperatörer & och avreferensoperatörer ».

Vi börjar med ett enkelt kodexempel:

```
int x = 42;
int y = 4x;
int z = 864;
x = y;
(x+1)+z;
y+=z;
```

Vika vänden har variablene x, y och z? Svarat är att värde på x är 43 men att värdena på y och z är okända¹.

Men låt oss belyja från bokser. För enkelsakens skull kan vi tänka oss att ett C-program har tillgång till två sorters minne: stackminne och heapminne. Stacken är det minnesutrymmet som används för att lagra värdena i lokala variabler i funktioner, så kallade automotiska variabler. Betrakta följande funktion:

```
void stupit(int value) {
    if (value) {
        int smaller = value - 1;
        stupit(smaller);
    }
}
```

Funktionen har två lokala variabler, value och smaller, båda av typen int. Låt oss anta att en int är 4 bytes. Det betyder att vi kommer att behöva 8 bytes på stacken för värdena i value och smaller².

Funktionen stupit är rekursiv; ett anrop stupit(3) kommer att leda till att funktionen anropar sig själv ytterligare tre gånger; varje funktion har en egen lokal variabel value vars värde är ett mindre än den anropande funktionens value-värde. Varje gång stupit anropas behövs ytterligare 8 bytes för att hålla värdena i dessa value och smaller-variabler. Vi säger att varje funktionsanrop leder till att ny stack-frame pushas på stacken som innehåller det minnesutrymmet

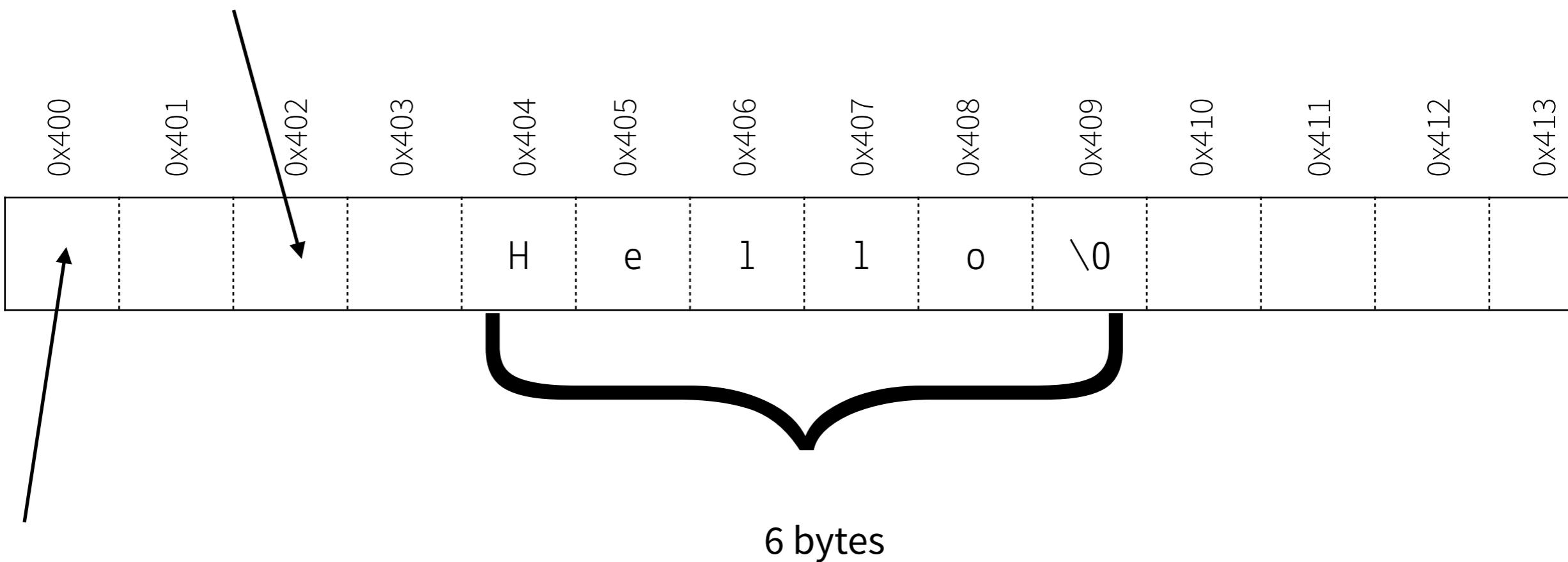
¹Dessutom så vet vi att oss värde på x är hälften x, så är värde på y + 4, givet att stacken på en pekare är 4 bytes.

²Det är en lätt återläggning – vissa andra data kan behövas, och smarta komplatorer gör optimiseringar som trots hot variabler som inte behövs, men ändå, anträder här. Vi berörer ihåll saken.



C:s minnesmodell

Varje ruta är en byte



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 = (uint8_t) atol(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>
```

1 byte

```
uint32_t fakultet(uint8_t f)
{
```

```
    if (f > 1)
    {
```

```
        return f * fakultet(f - 1);
    }
```

```
    else
    {
        return 1;
    }
}
```

? bytes

```
int main(int argc, char *argv[])
{
```

1 byte

```
    uint8_t n = (uint8_t) atol(argv[1]);
}
```

4 bytes

```
    uint32_t resultat = fakultet(n);
```

?? bytes

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

```
    return 0;
}
```

+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 = (uint8_t) atol(argv[1]);
    uint32_t resultat = fakultet(n);

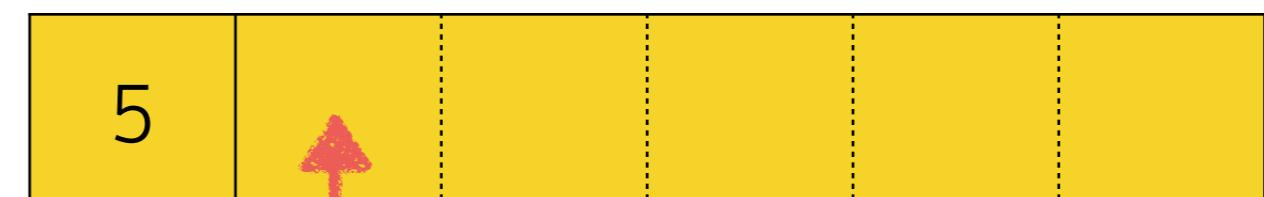
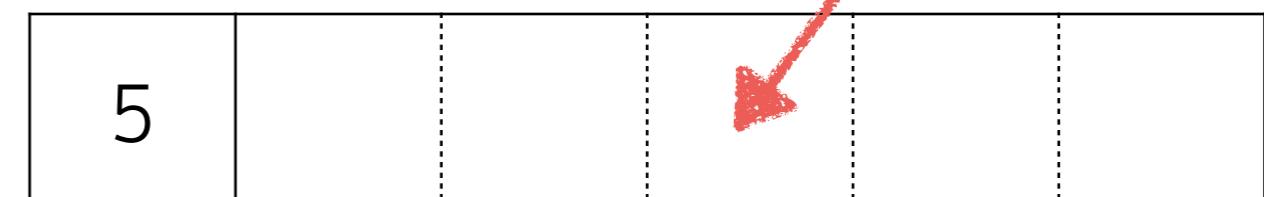
    printf("%d! = %d\n", n, resultat);

    return 0;
}

```

f <return value>

fakultet:s
stackframe



main:s stackframe

```

#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 = (uint8_t) atol(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;
    }
}

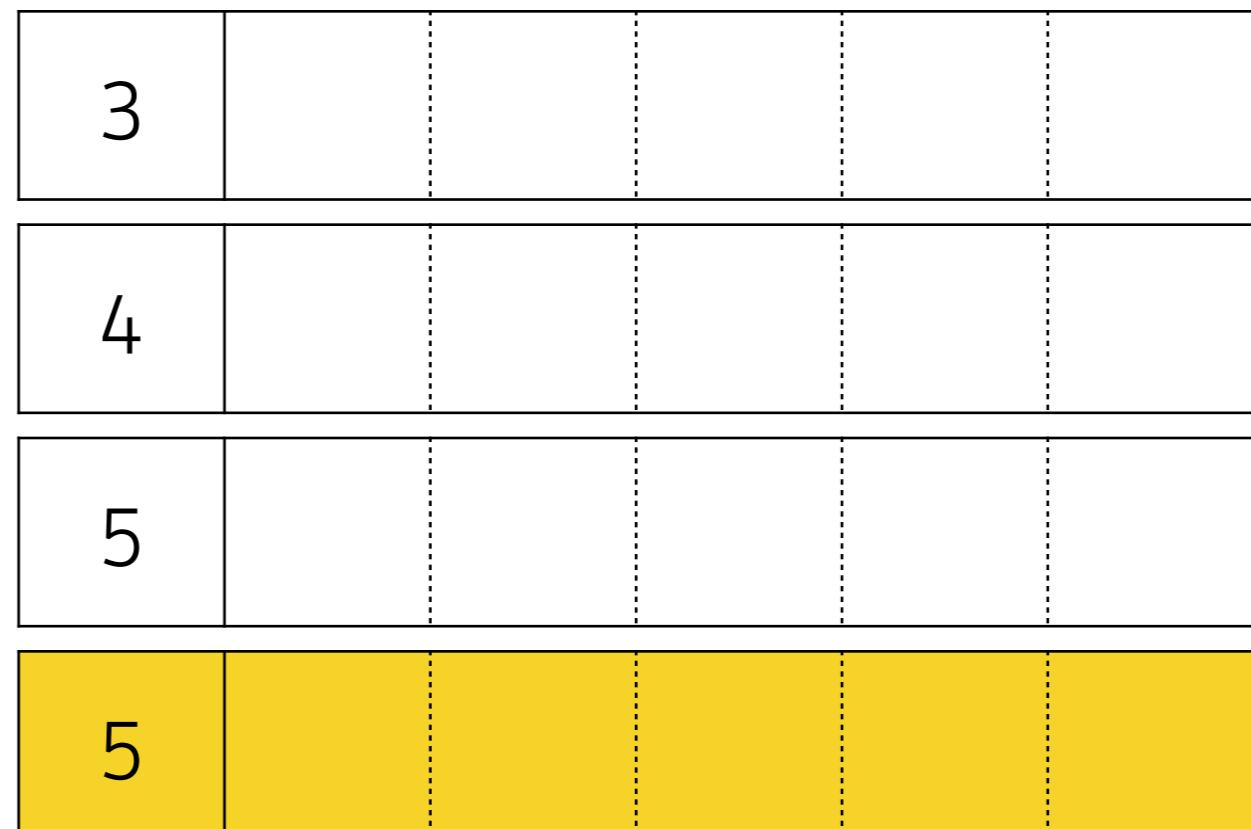
int main(int argc, char *argv[])
{
    uint8_t n = (uint8_t) atol(argv[1]);
    uint32_t resultat = fakultet(n);

    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 = (uint8_t) atol(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 = (uint8_t) atol(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 = (uint8_t) atol(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 = (uint8_t) atol(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 = (uint8_t) atol(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 = (uint8_t) atol(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						

```

#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 = (uint8_t) atol(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						

```

#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 = (uint8_t) atol(argv[1]);
    uint32_t resultat = fakultet(n);

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

    return 0;
}

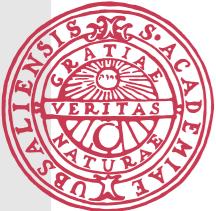
```

f <return value>

1						1
2						2
3						6
4						24
5						120
5						

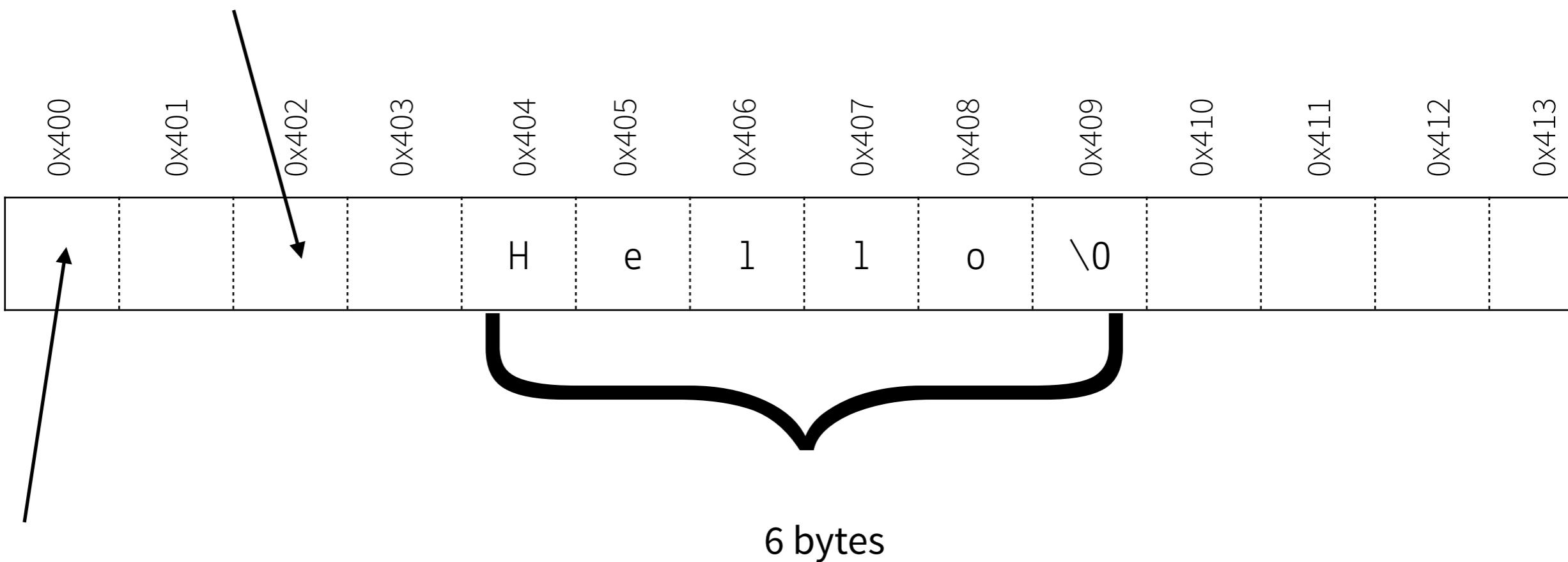


“Pekare”



C:s minnesmodell

Varje ruta är en byte



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

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



```
int x = 42;    // x innehåller ett heltal  
  
int *p;        // p innehåller en adress till en  
                  // plats i minnet där det finns ett  
                  // heltal t.ex. 0x404
```



```
int x = 42;    // x innehåller ett heltal  
  
int *p;        // p innehåller en adress till en  
                  // plats i minnet där det finns ett  
                  // heltal  
  
p = &x;         // uppdatera pekaren p
```



```
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
```



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



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



”Nullpekar” (1/2)

- ”I call it my billion-dollar mistake. It was the invention of the null reference in 1965” — Tony Hoare

Syntax: NULL

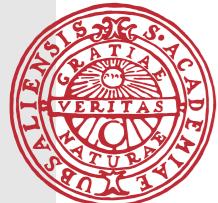
Semantik: ”variabeln pekar inte på någonting”

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



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

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



NULL POINTER DEREFERENCE

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

```
int y = *x,
```



Ny typ: **void ***

Tas upp mer senare

- En pekare till något av okänd typ

C tillåter **inte** att den avrefereras

Användbar för att skapa t.ex. generella lagringsklasser (jmf. inlupp 2, lagerh.)

```
int x;  
int *y = &x;  
void *z = y;
```

```
*y = 42; // ok  
*z = 42; // kompilerar ej
```

```
int a = *y; // ok  
int b = *z; // kompilerar ej
```

```
int *c = (int *)z;  
*c = 42; // ok
```

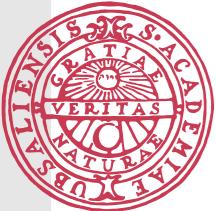
Pekare: recap

`&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 pekarer



```

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

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

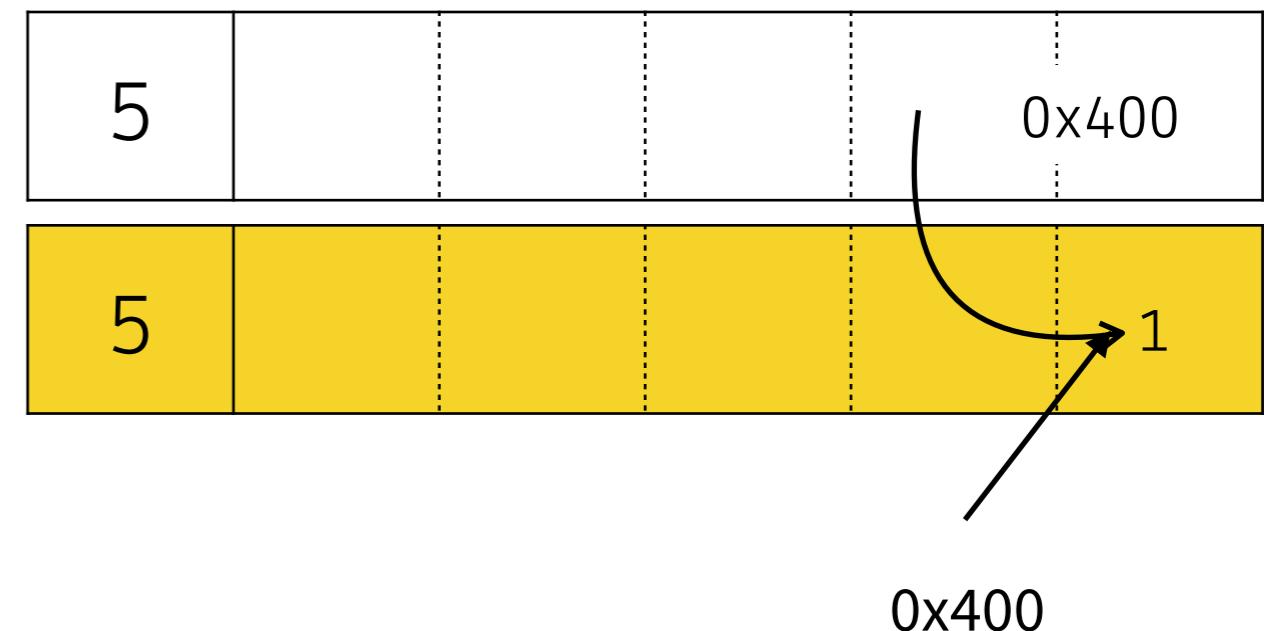
int main(int argc, char *argv[])
{
    uint8_t n = (uint8_t) atol(argv[1]);
    uint32_t resultat = 1;
    fakultet(n, &resultat);

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

    return 0;
}

```

f <return value>



```

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

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

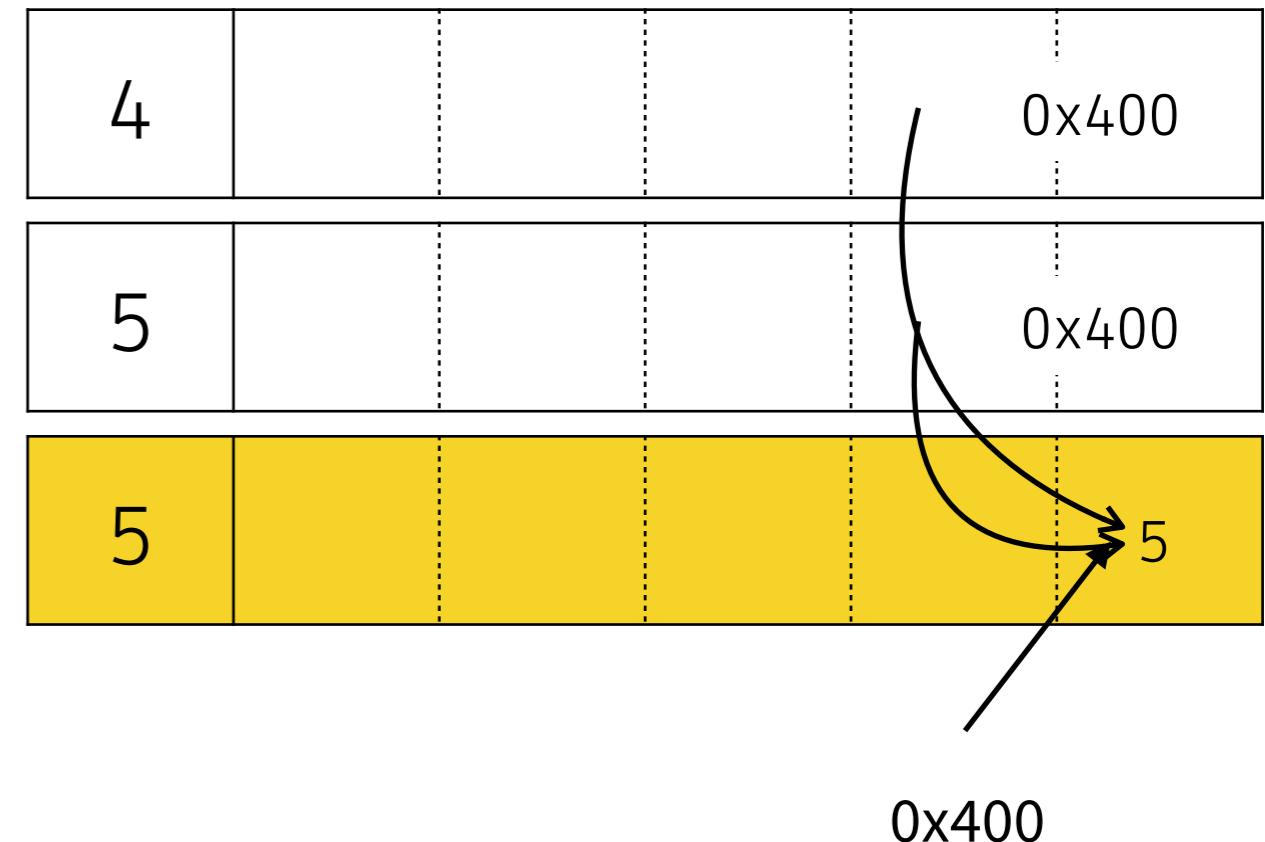
int main(int argc, char *argv[])
{
    uint8_t n = (uint8_t) atol(argv[1]);
    uint32_t resultat = 1;
    fakultet(n, &resultat);

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

    return 0;
}

```

f <return value>



```

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

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

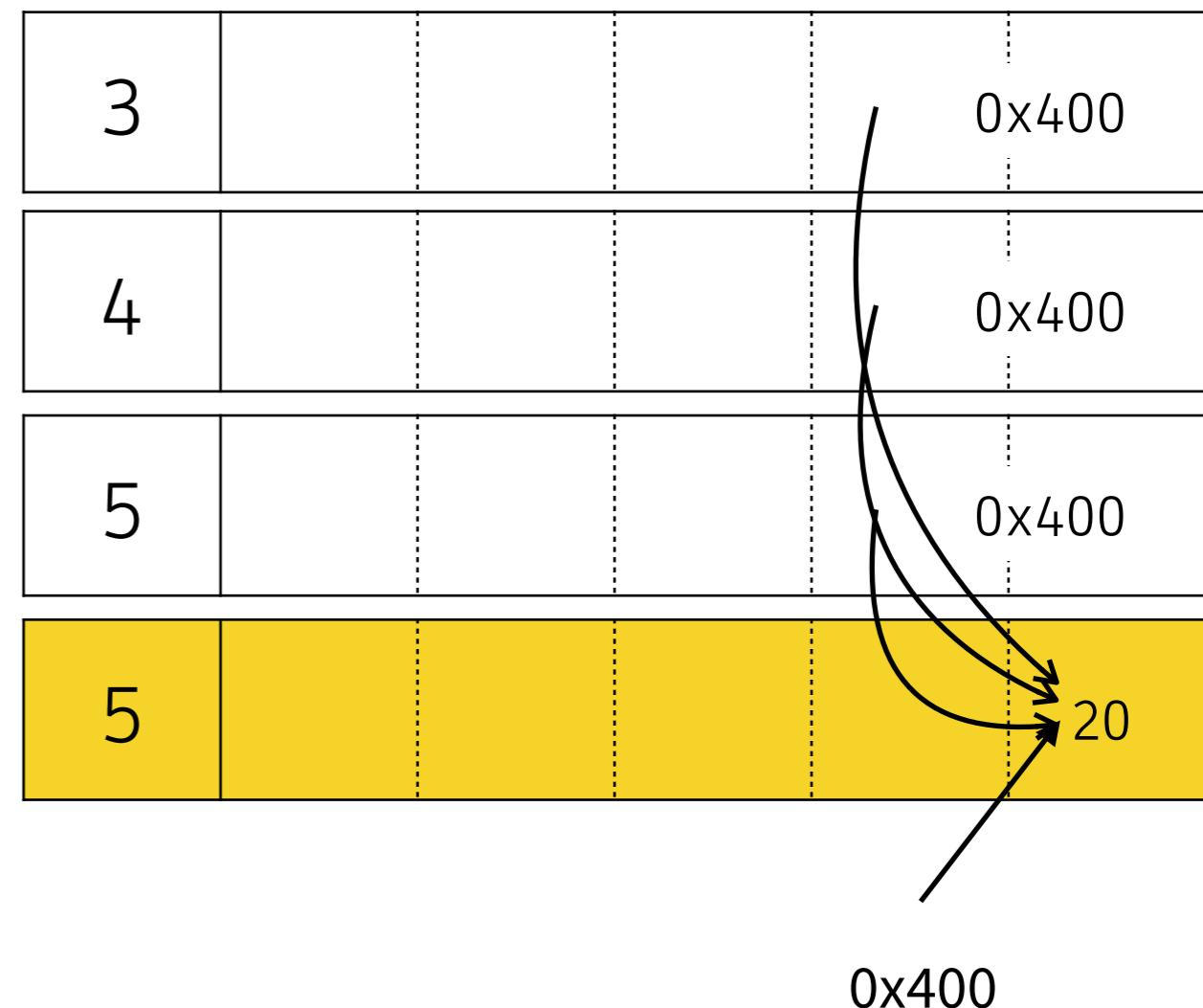
int main(int argc, char *argv[])
{
    uint8_t n = (uint8_t) atol(argv[1]);
    uint32_t resultat = 1;
    fakultet(n, &resultat);

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

    return 0;
}

```

f <return value>



```

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

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

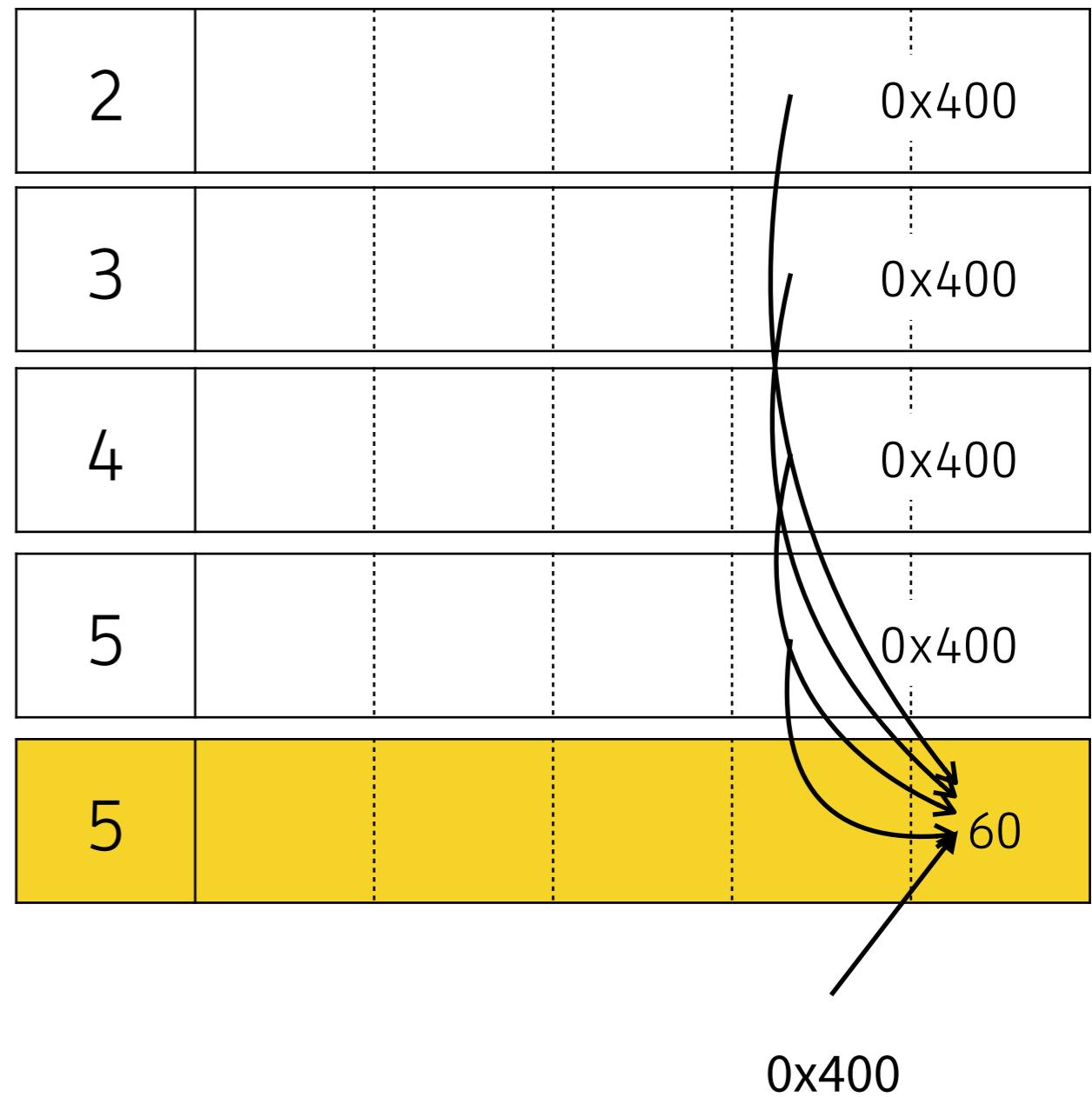
int main(int argc, char *argv[])
{
    uint8_t n = (uint8_t) atol(argv[1]);
    uint32_t resultat = 1;
    fakultet(n, &resultat);

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

    return 0;
}

```

f <return value>



```

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

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

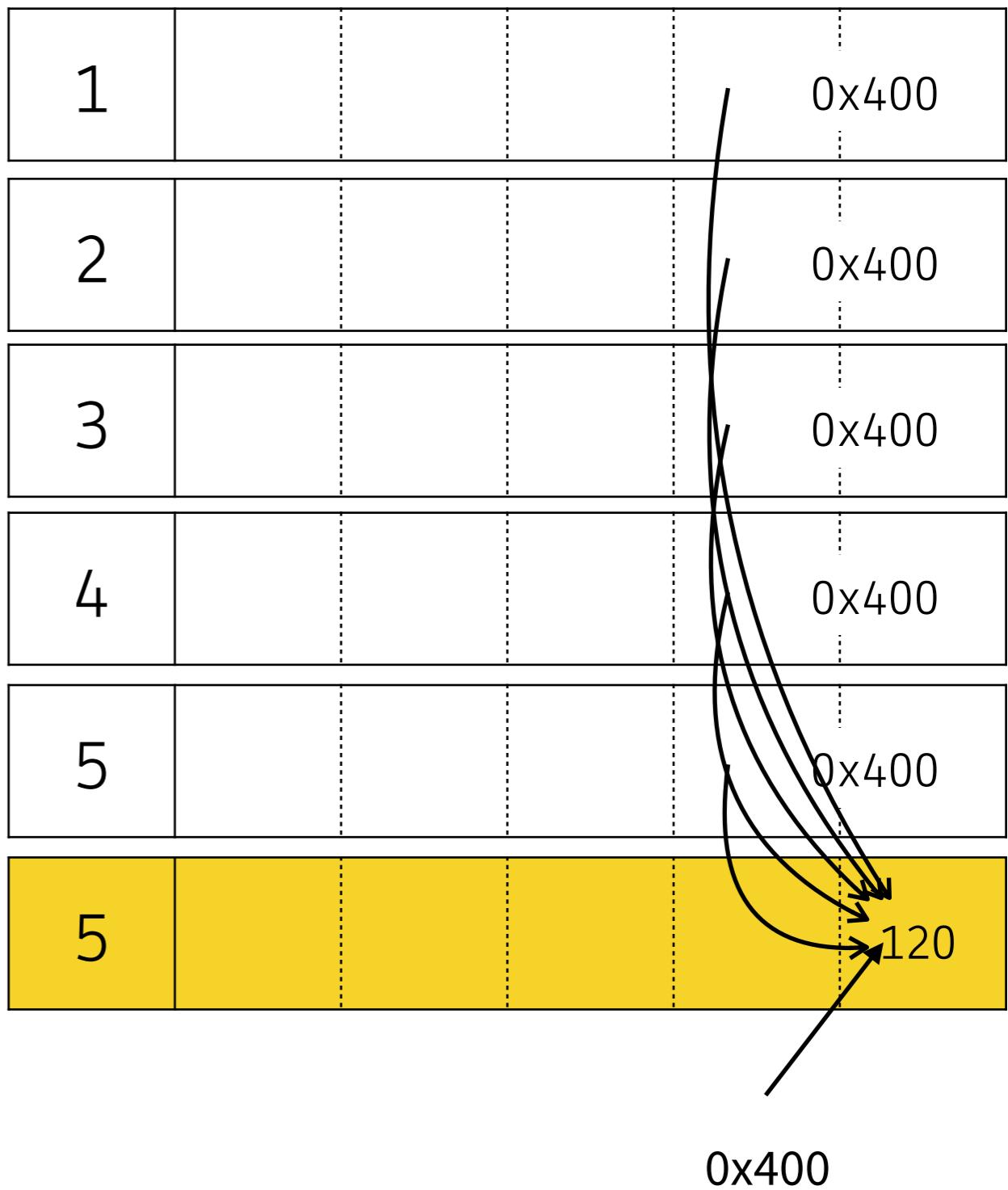
int main(int argc, char *argv[])
{
    uint8_t n = (uint8_t) atol(argv[1]);
    uint32_t resultat = 1;
    fakultet(n, &resultat);

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

    return 0;
}

```

f <return value>



```

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

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

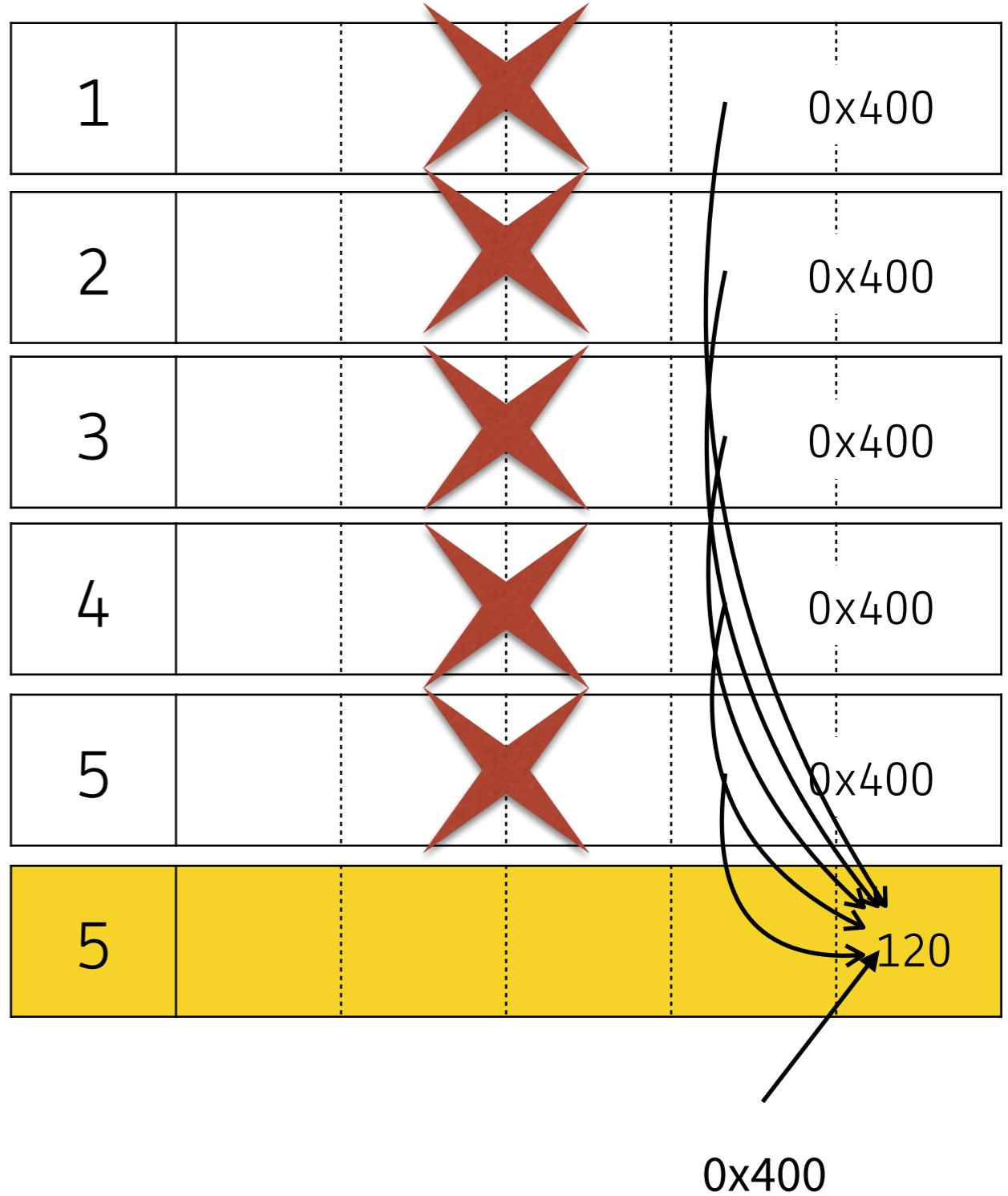
int main(int argc, char *argv[])
{
    uint8_t n = (uint8_t) atol(argv[1]);
    uint32_t resultat = 1;
    fakultet(n, &resultat);

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

    return 0;
}

```

f <return value>



```
#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 = (uint8_t) atol(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

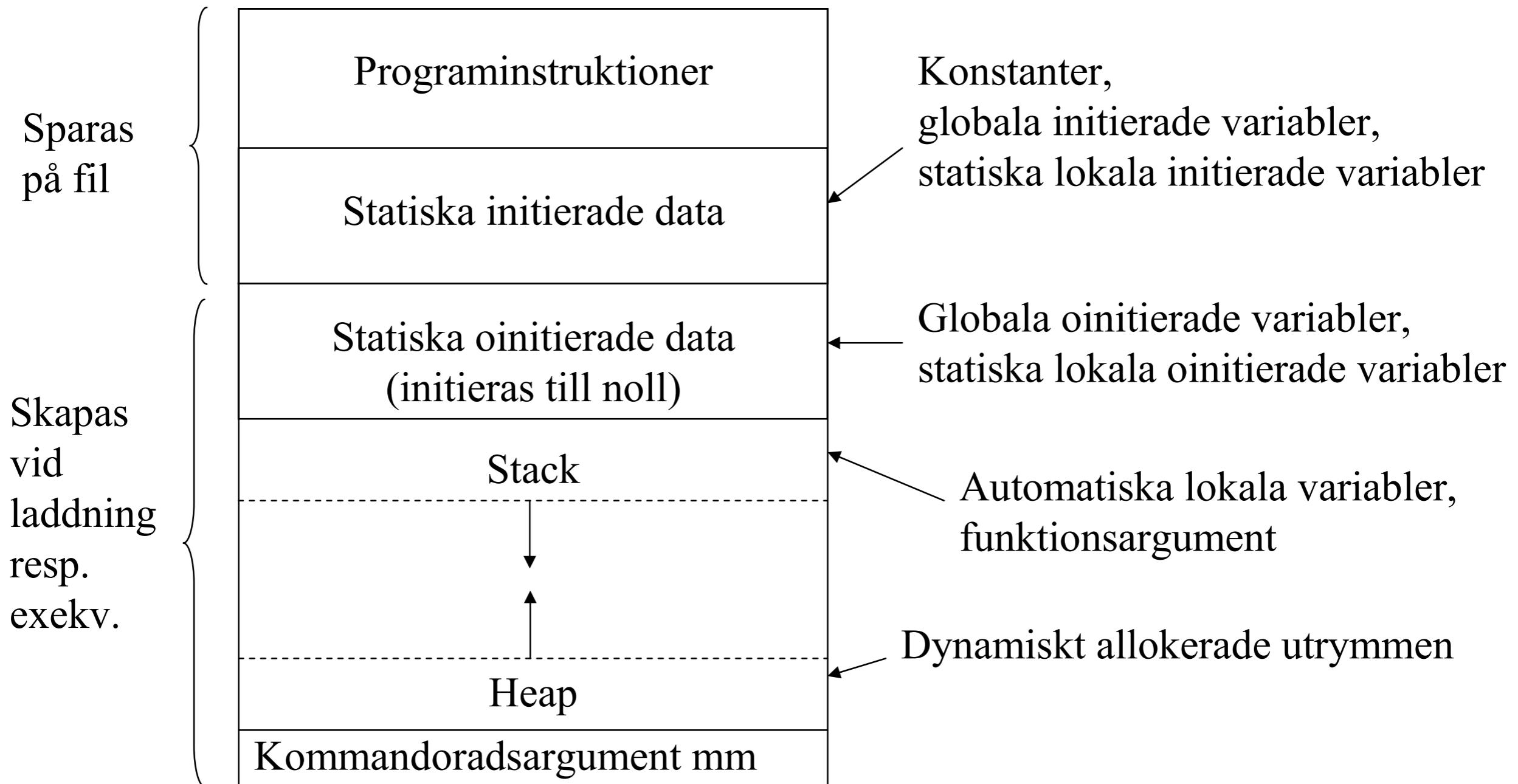
Håller ”kortlivat data” (knutet till funktionens livslängd)

Varje funktionsanrop ger upphov till en ny ”stack frame”
(not. parameteröverföring sker i regel mha register)

Allokeras och avallokeras automatiskt

- Extremt effektiv allokeringsmetod (”bump pointer”)
- Stacken är en del av minnet — alla variabler på stacken har en adress och kan pekas ut av pekare

Var lagras data?



Heopen

Läs Addendum om stack och heap



Ett enkelt stackexempel

Skrivet saso-det har lilla hället är att driva hem några posturer kring skillnaderna mellan stack och heap, adressstyrningsepateren & och avreferensningsoperatoren ».

Vi börjar med ett enkelt kodexempel:

```
int x = 42;  
int y = &x;  
int z = &y[0];  
x = y;  
(x)+=1;  
y+=2;
```

Vika vänden har variablene x, y och z? Svarat är att värde på x är

43 men att värdena på y och z är okända¹.

Men låt oss belyja från belyjan. För enkeltetens skull kan vi tänka oss att ett C-program har tillgång till två sorters minne: stackminne och heapminne. Stacken är det minnesutrymmet som används för att lagra värdena i lokala variabler i funktioner, så kallade automotiska variabler. Betrakta följande funktion:

```
void stupit(int value) {  
    if (value) {  
        int smaller = value - 1;  
        stupit(smaller);  
    }  
}
```

Funktionen har två lokala variabler, value och smaller, båda av typen int. Låt oss anta att en int är 4 bytes. Det betyder att vi kommer att behöva 8 bytes på stacken för värdena i value och smaller².

Funktionen stupit är rekursiv; ett anrop stupit(3) kommer att leda till att funktionen anropar sig själv ytterligare tre gånger; varje funktion har en egen lokal variabel value vars värde är ett mindre än den anropande funktionens value-värde. Varje gång stupit anropas behövs ytterligare 8 bytes för att hålla värdena i dessa value och smaller-variabler. Vi säger att varje funktionsanrop leder till att ny stack-frame pushas på stacken som innehåller det minnesutrymmet

¹Dessutom så vet vi att om värde på z är hället s, så är värde på y + 4, givet att stacken på en pekare är 4 bytes.

²Det är en lång förmodning – vissa andra data kan behövas, och smarta komplatorer gör optimiseringar som trots hot variabler inte räcker ihop, men tan, snäller här. Vi berör ihåd detta.



Heapen

- Till för lagring av ”långlivat” data
- Hanteras manuellt

För varje data som skall lagras på heapen måste man explicit be om ett motsvarande utrymme mha `malloc` (äv. `realloc` och `calloc`)

När man är färdig med data måste man explicit returnera det, annars läcker programmet minne mha funktionen `free`

- Heapen bara tillgänglig via pekare

Exempel: funktionen strdup()

```
char *strdup(const char *str)
{
    char *dup = malloc(strlen(str)+1);
    char *cursor = dup;

    while (*str)
    {
        *cursor = *str;
        ++str;
        ++cursor;
    }
    *cursor = '\0';

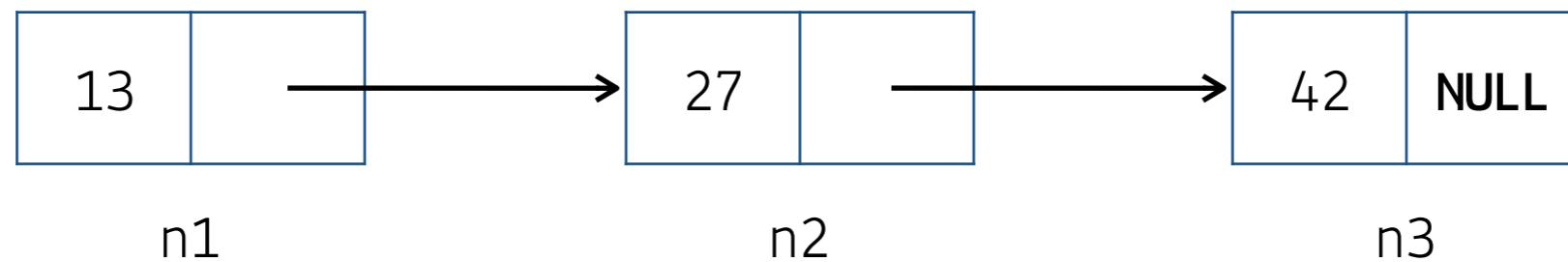
    return dup;
}
```

Dynamiska strukturer

- Heopen 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!

Pekare och länkade strukturer [På stacken]



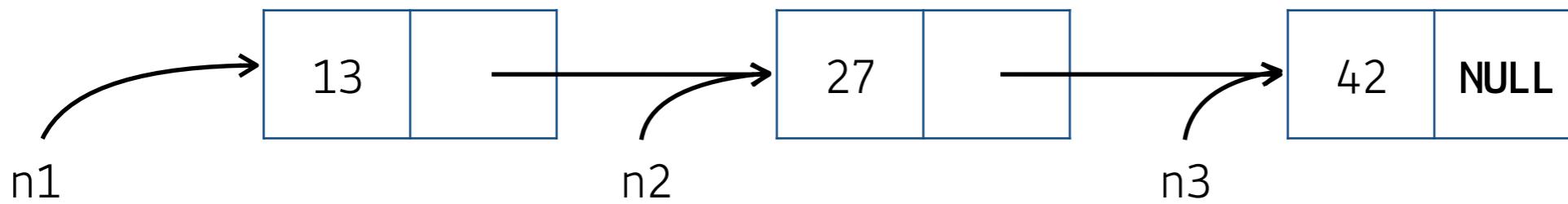
```
struct node n1;
struct node n2;
struct node n3;
```

```
n1 = (struct node) { .number = 13, .next = &n2 };
n2 = (struct node) { .number = 27, .next = &n3 };
n3 = (struct node) { .number = 42, .next = NULL };
```

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



Pekare och länkade strukturer [På heapen]



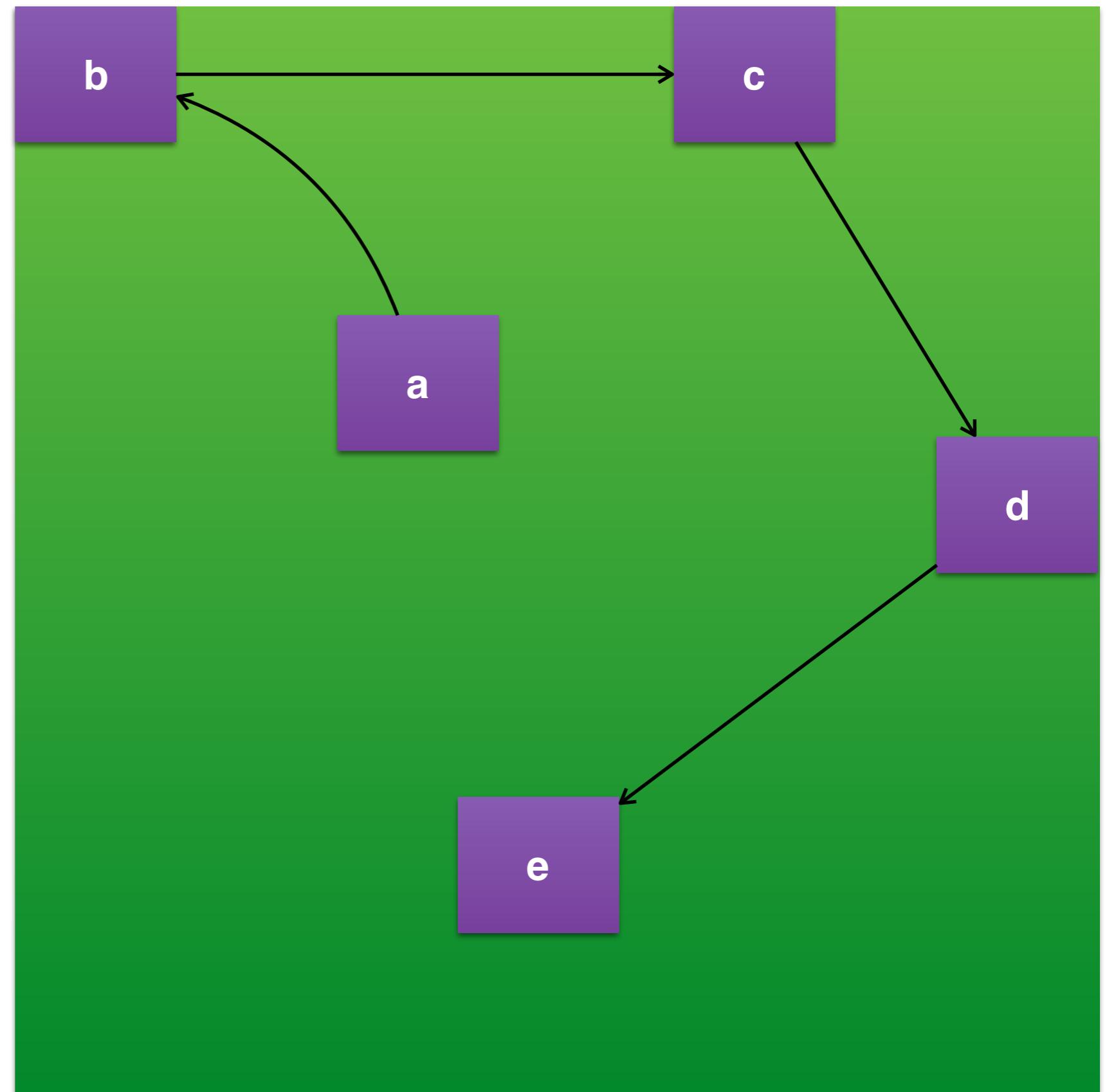
```
struct node *n1 = malloc(sizeof(struct node));
struct node *n2 = malloc(sizeof(struct node));
struct node *n3 = malloc(sizeof(struct node));

*n1 = (struct node) { .number = 13, .next = n2 };
*n2 = (struct node) { .number = 27, .next = n3 };
*n3 = (struct node) { .number = 42, .next = NULL };
```



En ”länkad” lista

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



Avreferera pekare

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



Följ pekaren



Läs next-posten
i strukten



Båda i samma
operator



```
struct tree_node
{
    node_t *left;
    node_t *right;
    int key;
    data_t *data;
};
```

Att skapa noder i ett träd

```
node_t node_new(int key, data_t *data)
{
    node_t *n = malloc(sizeof(node_t));
    if (n)
    {
        n->left = NULL;
        n->right = NULL;
        n->key = key;
        n->data = data;
    }
    return n;
}
```



Ta bort noder ur ett träd

```
void node_deallocate1(node_t *n)
{
    if (n->left) free(n->left);
    if (n->right) free(n->right);
    free(n);
}
```

```
void node_deallocate2(node_t *n)
{
    if (n->left) free(n->left);
    if (n->right) free(n->right);
    free(n->data);
    free(n);
}
```

```
data_t *node_deallocate3(node_t *n)
{
    if (n->left) free(n->left);
    if (n->right) free(n->right);
    free(n);
    return n->data; // BOOOOM!!!!
}
```

```
data_t *node_deallocate4(node_t *n)
{
    if (n->left) free(n->left);
    if (n->right) free(n->right);
    data_t *tmp = n->data;
    free(n);
    return tmp;
}
```



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?

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

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

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

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

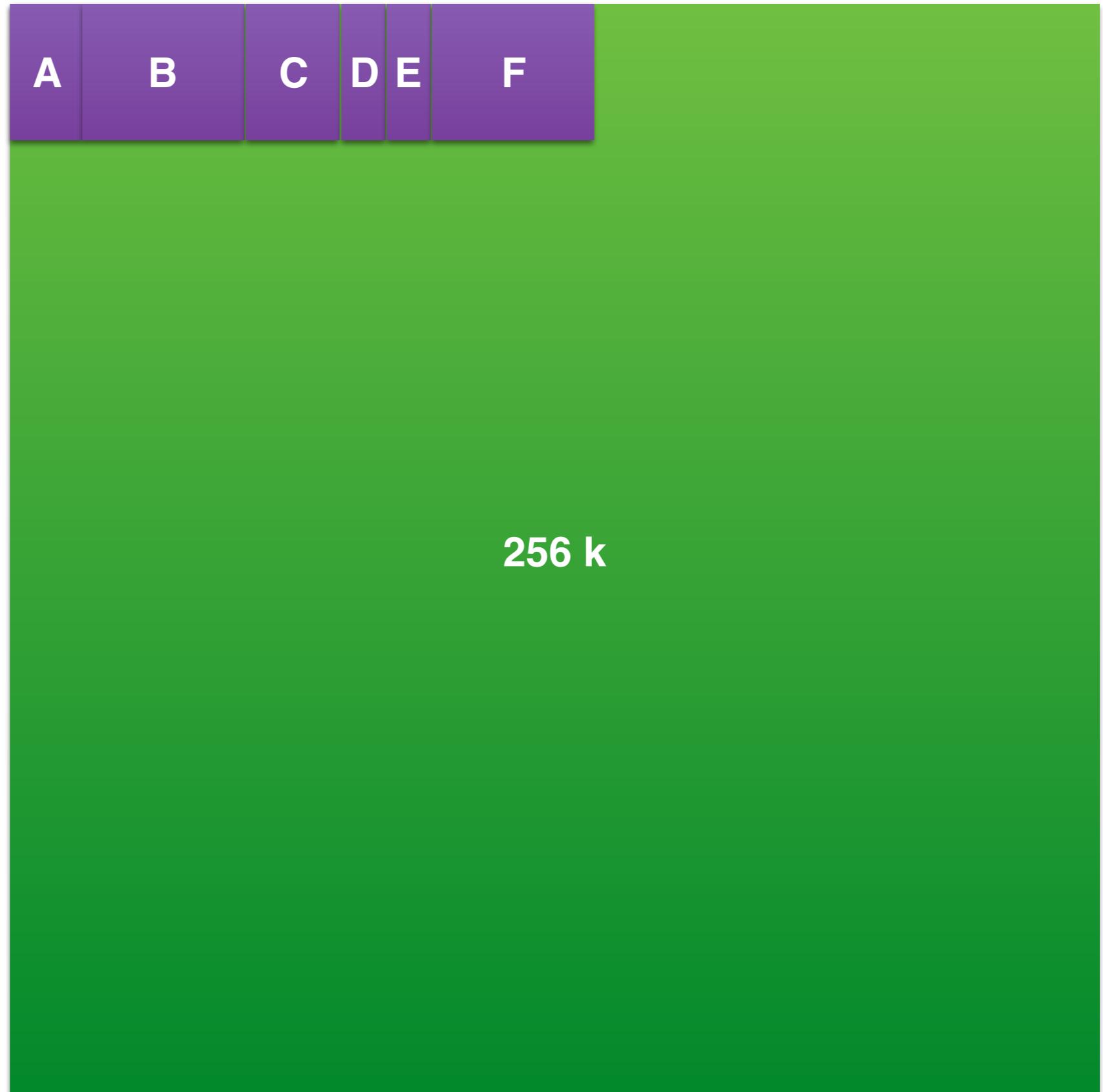
```
e = malloc(E);
```

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

```
free(a);
```

```
free(c);
```

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



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

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

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

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

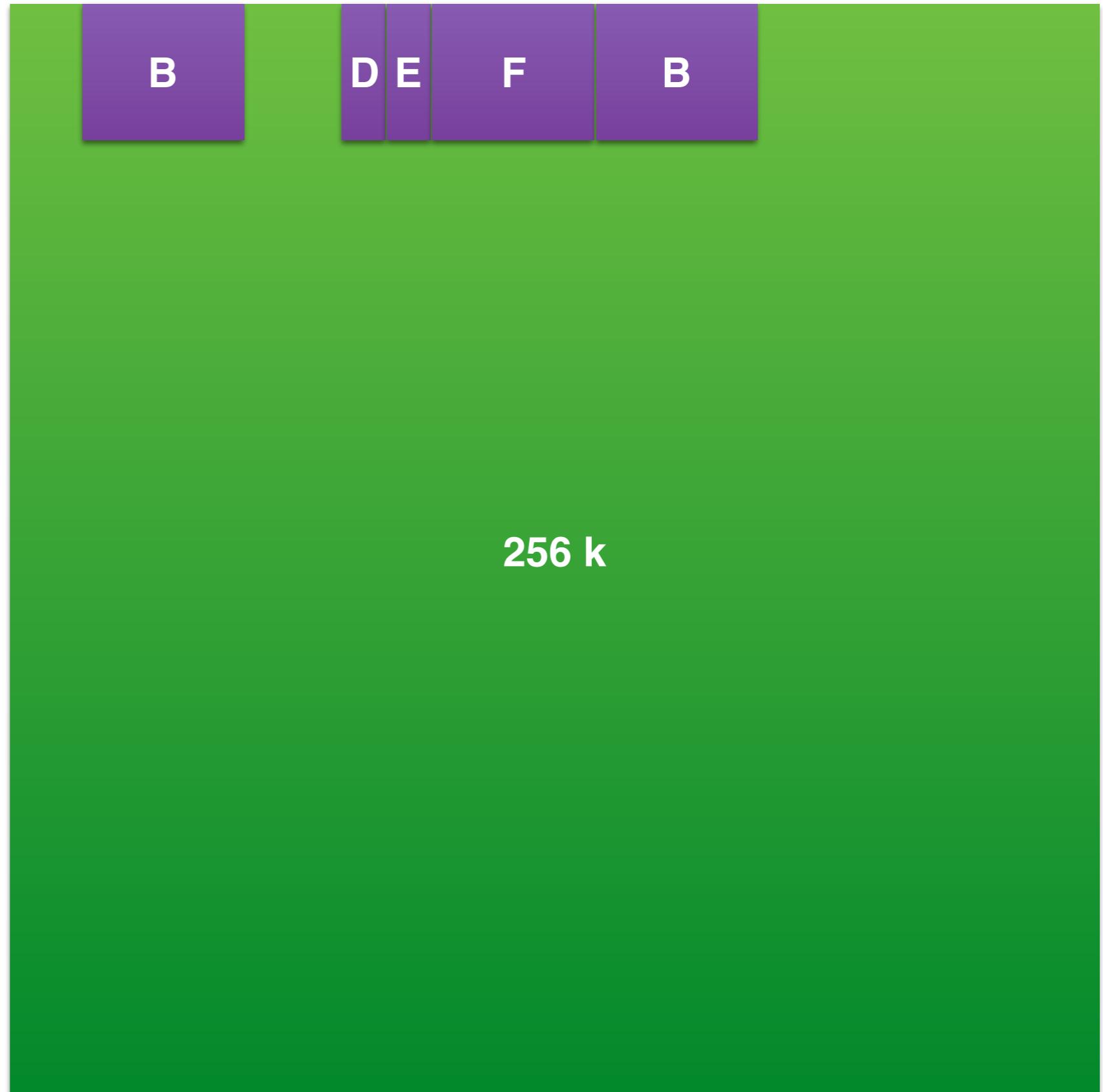
```
e = malloc(E);
```

```
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 samman-
hängande (konsekutiva;
contiguous)*

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

B

D E

F

B

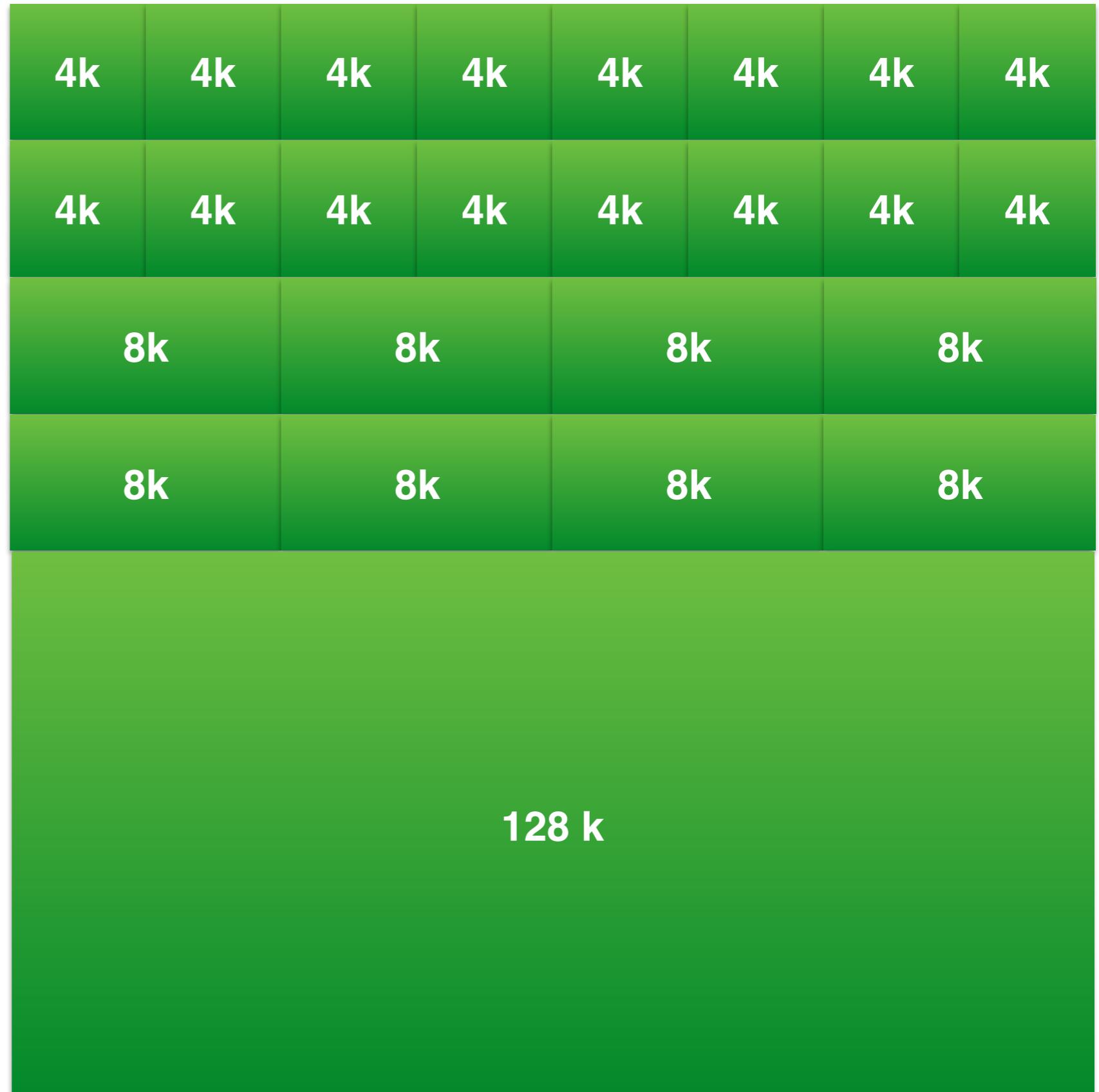
256 k



Bucket Allocation

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

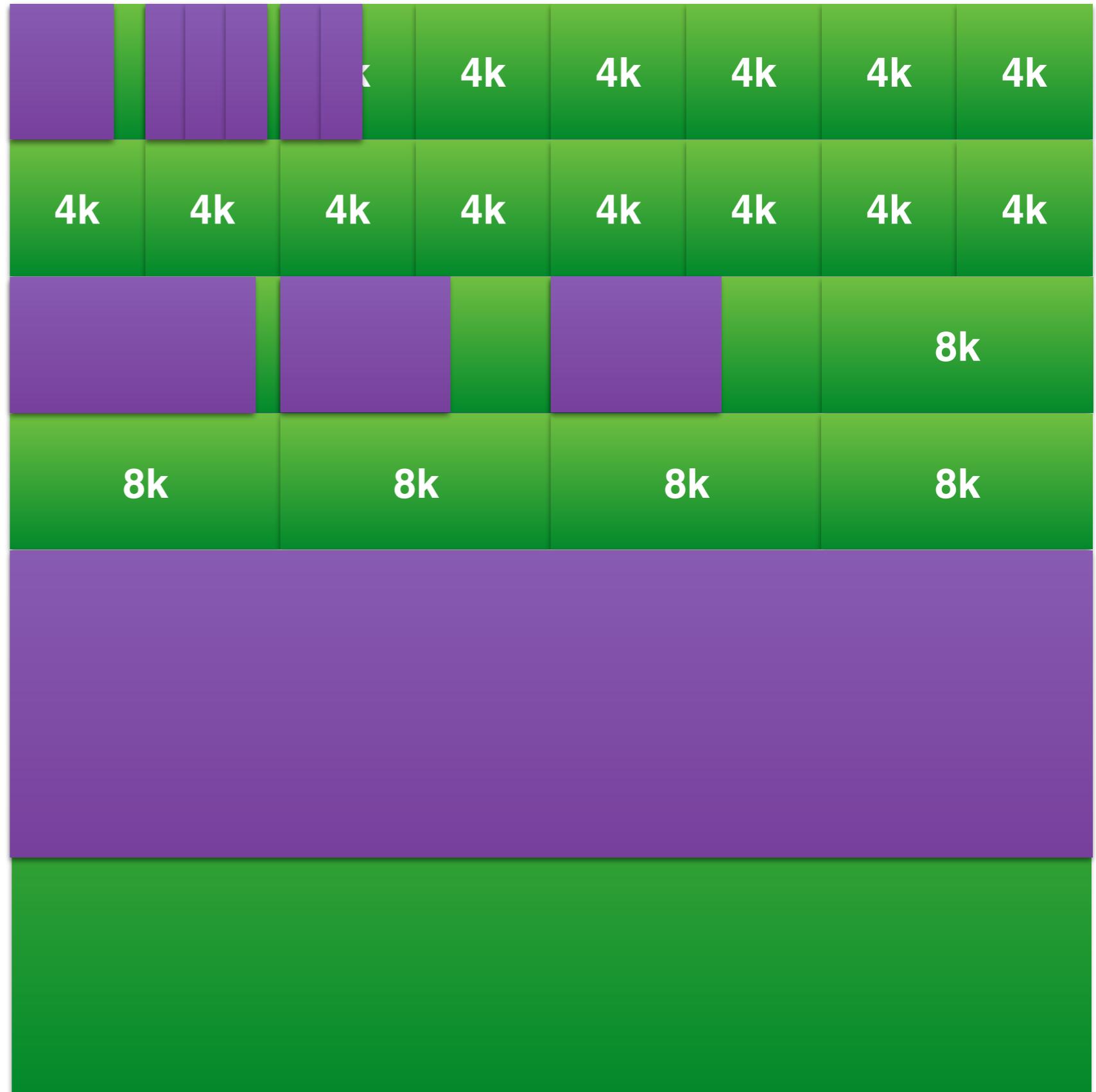
*Vad får detta för effekt
map fragmentering?*



Bucket Allocation

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

*Vad får detta för effekt
map fragmentering?*



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 allokerar nog med yta
- Måste själv anropa free i rätt tid
- Se valgrind för verktygsstöd för att hantera minne
- ”malloc är inte magisk”

Sammanfattning

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

Extramaterial

Hitta felen!



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;
}
```



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;
}
```

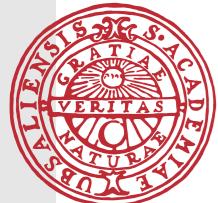
1. Vi flyttar pekaren i minnet

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;
}
```



2. Men vad händer om $\text{fst} == \text{snd}$?

```
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

- Kraftfullt!
- Livsfarligt!
- Fundamentalt problem i programspråksfältet

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



Vad skrivs ut av
detta program?

OBS! Ej tillräcklig information
på denna bild för att kunna
besvara den frågan!

```
#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(void)
{
    int8_t even[] = { 2, 4, 6, 8 };
    int8_t odd[] = { 1, 3, 5, 7 };

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

    for (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(void)
{
    int8_t even[] = { 2, 4, 6, 8 };
    int8_t odd[] = { 1, 3, 5, 7 };

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

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

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

1. Samma kod
som "sist"

2. even och odd
är allokerade på
stacken!

Hitta felet!

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.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(void)
{
    int8_t *ns = malloc(4 * sizeof(int8_t));
    ns[0] = 1; ns[1] = 3; ns[2] = 5; ns[3] = 7;

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

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

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



add
anropas
"med
samma
array"

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.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(void)
{
    int8_t *ns = malloc(4 * sizeof(int8_t));
    ns[0] = 1; ns[1] = 3; ns[2] = 5; ns[3] = 7;

    int8_t *sum = add(4, ns, ns);
    
    for (i = 0; i < 4; ++i)
    {
        printf("%d ", sum[i]);
    }

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

