Imperativ och objektorienterad programmeringsmetodik

Föreläsning 3 av många

Tobias Wrigstad

Dagens agenda

Minne

- Stack
- Heap
- ...

Pekare & pekarvariabler

Dynamisk allokering och avallokering av minne

Introduktion till länkade strukturer

Allt ett program gör kräver minne

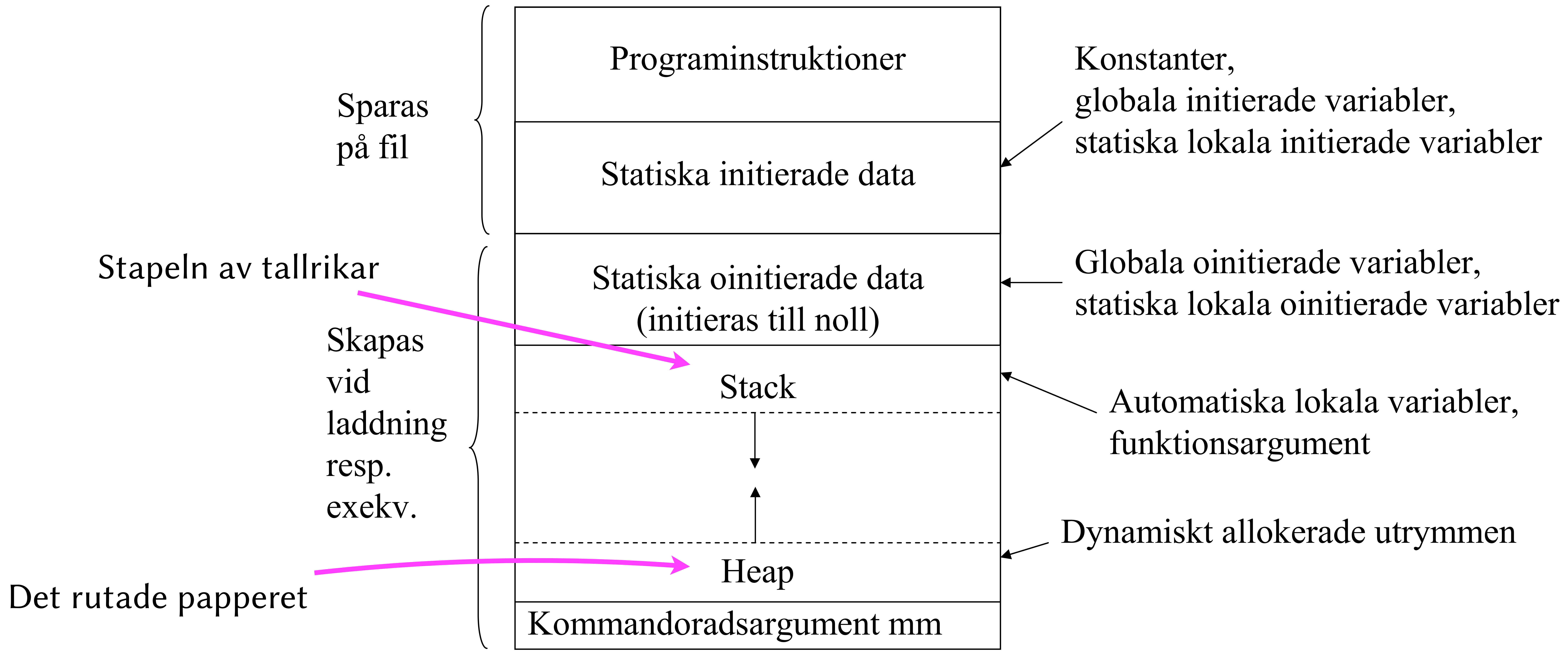
- Koden för en funktion behöver sparas någonstans
- Varje funktion behöver minne för att spara sina lokala variabler

  Funktioner som inte är svansrekursiva kräver minne linjärt mot rekursionsdjupet

- ...

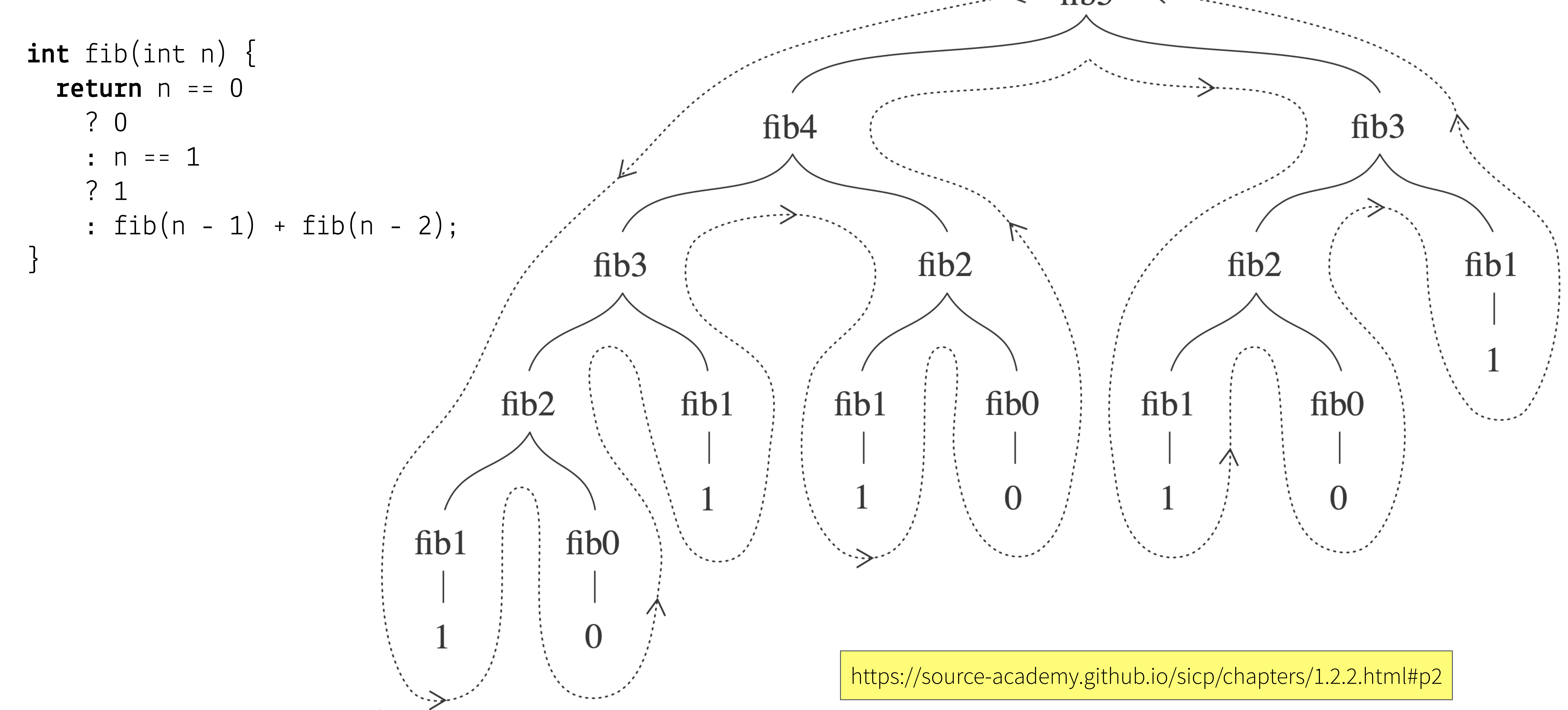
  Om min dator arbetsminne är 16 GB kan jag inte lagra åldern på varje person på jorden i en `int` (men väl i en `char`)

Var lagras data i ett C-program?



Exempelprogram: rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```



Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

argc = … argv = …

Stackanvändning i rekursiv fibonacci

```c
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 5	
argc = …	argv = …

Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 4
n = 5
argc = … argv = …

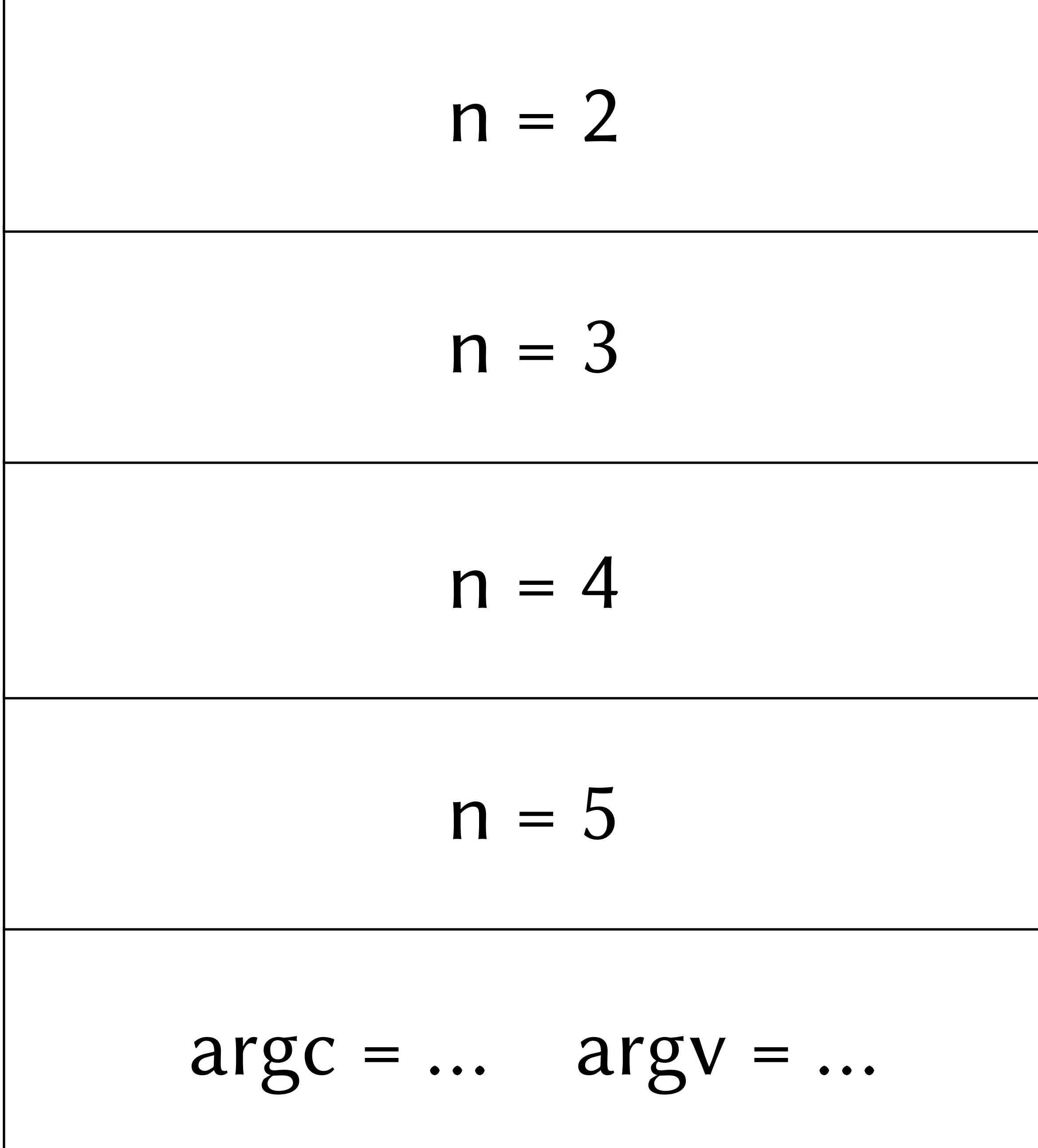
Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 3
n = 4
n = 5
argc = … argv = …

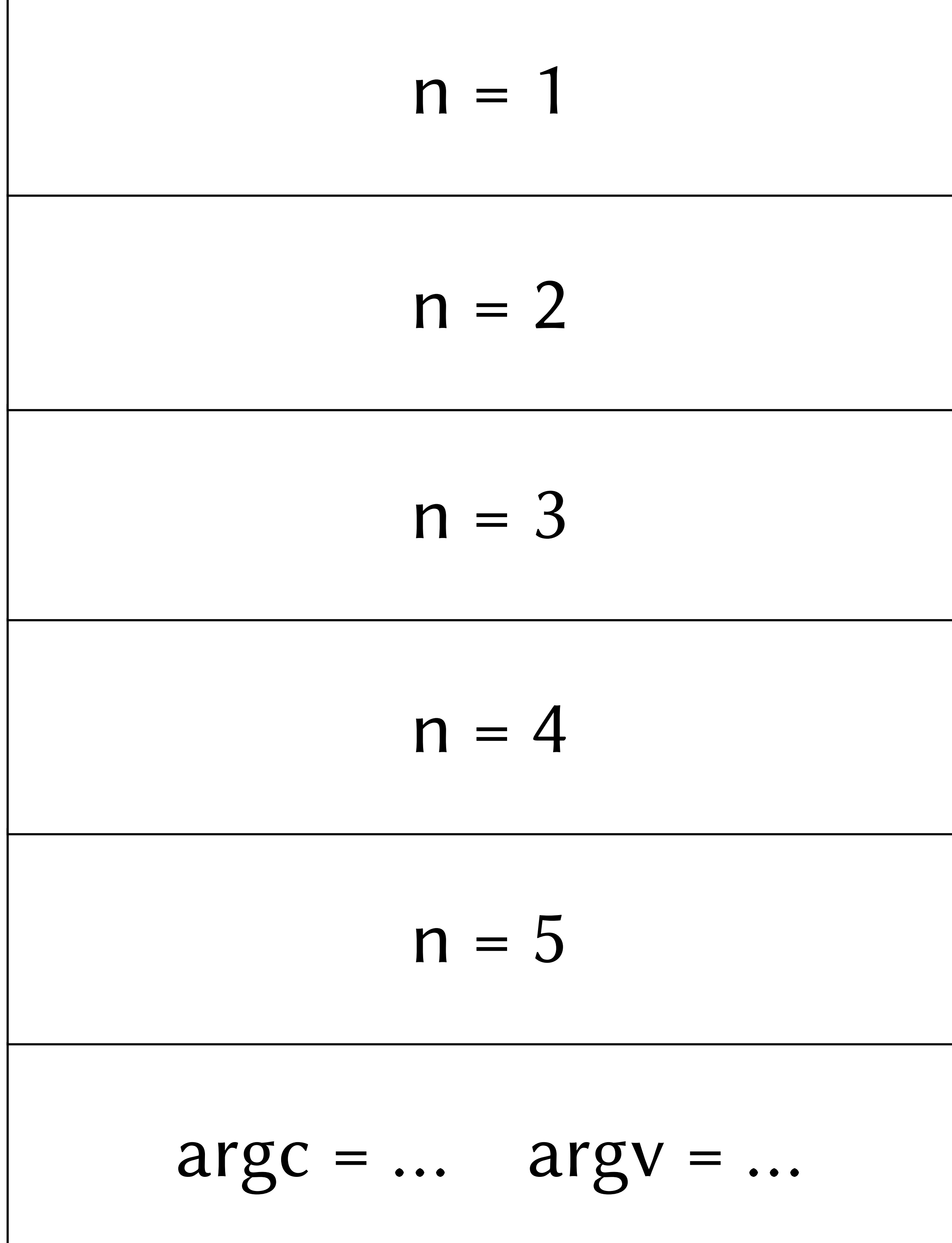
Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```



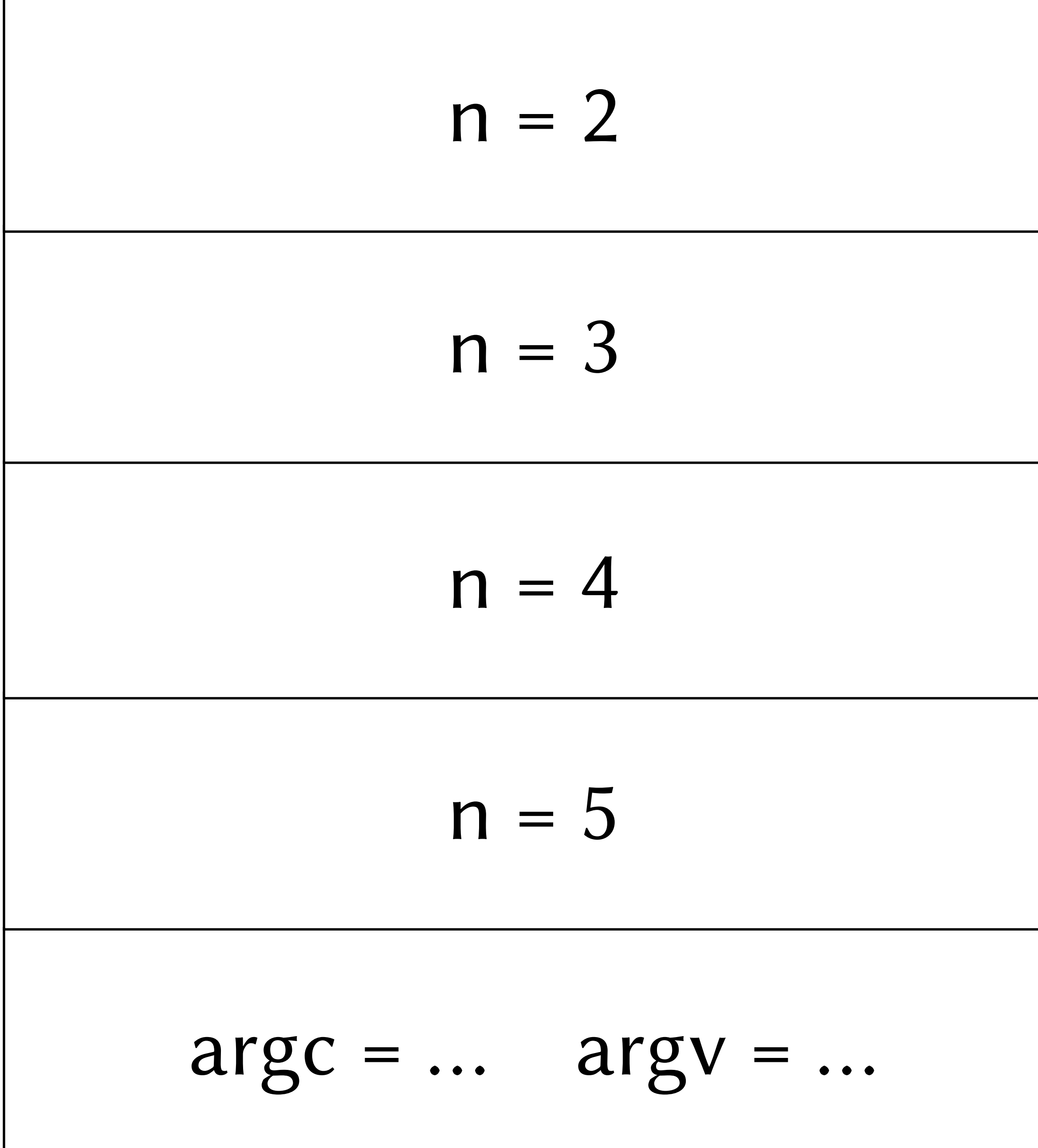
Stackanvändning i rekursiv fibonacci

```c
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```



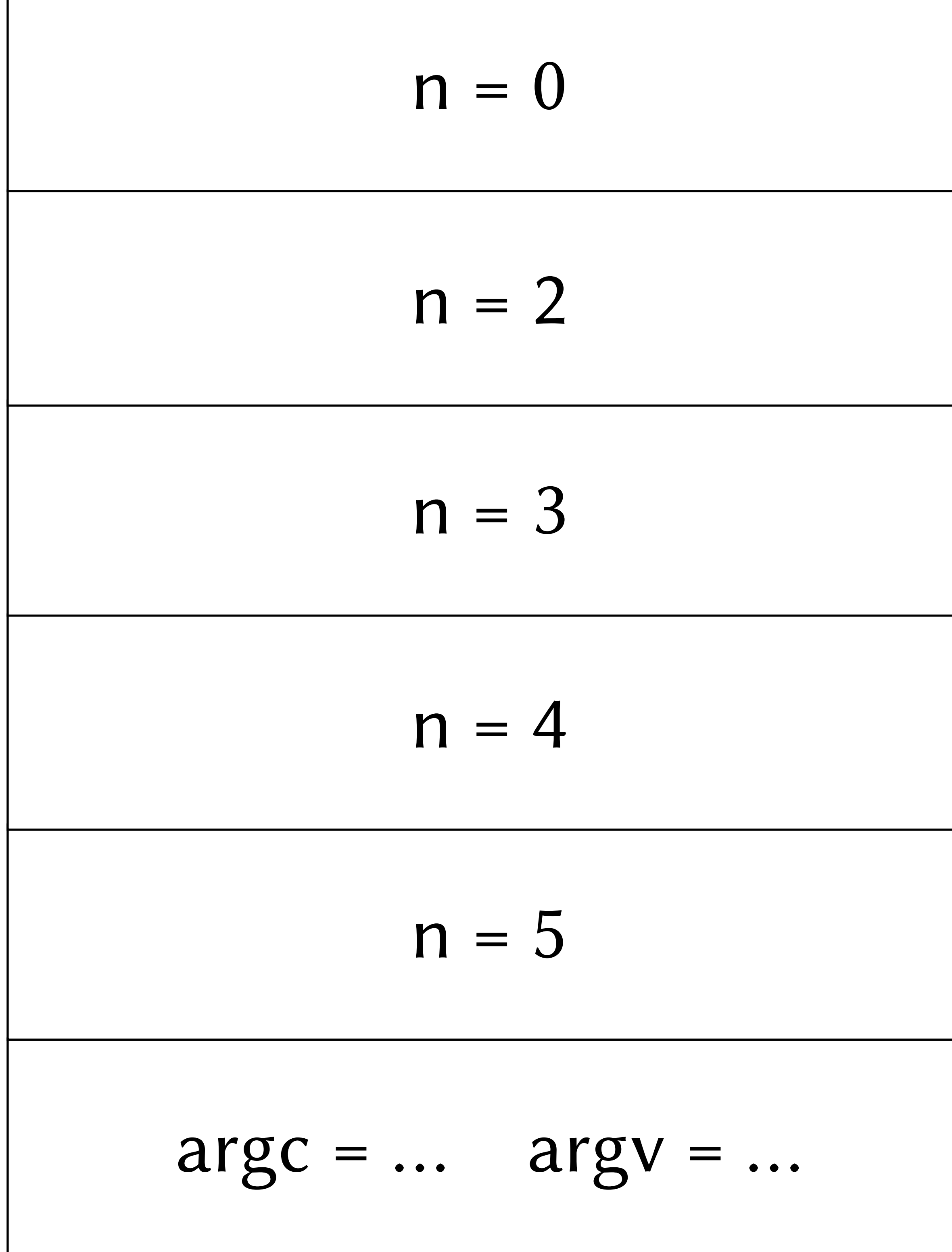
Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```



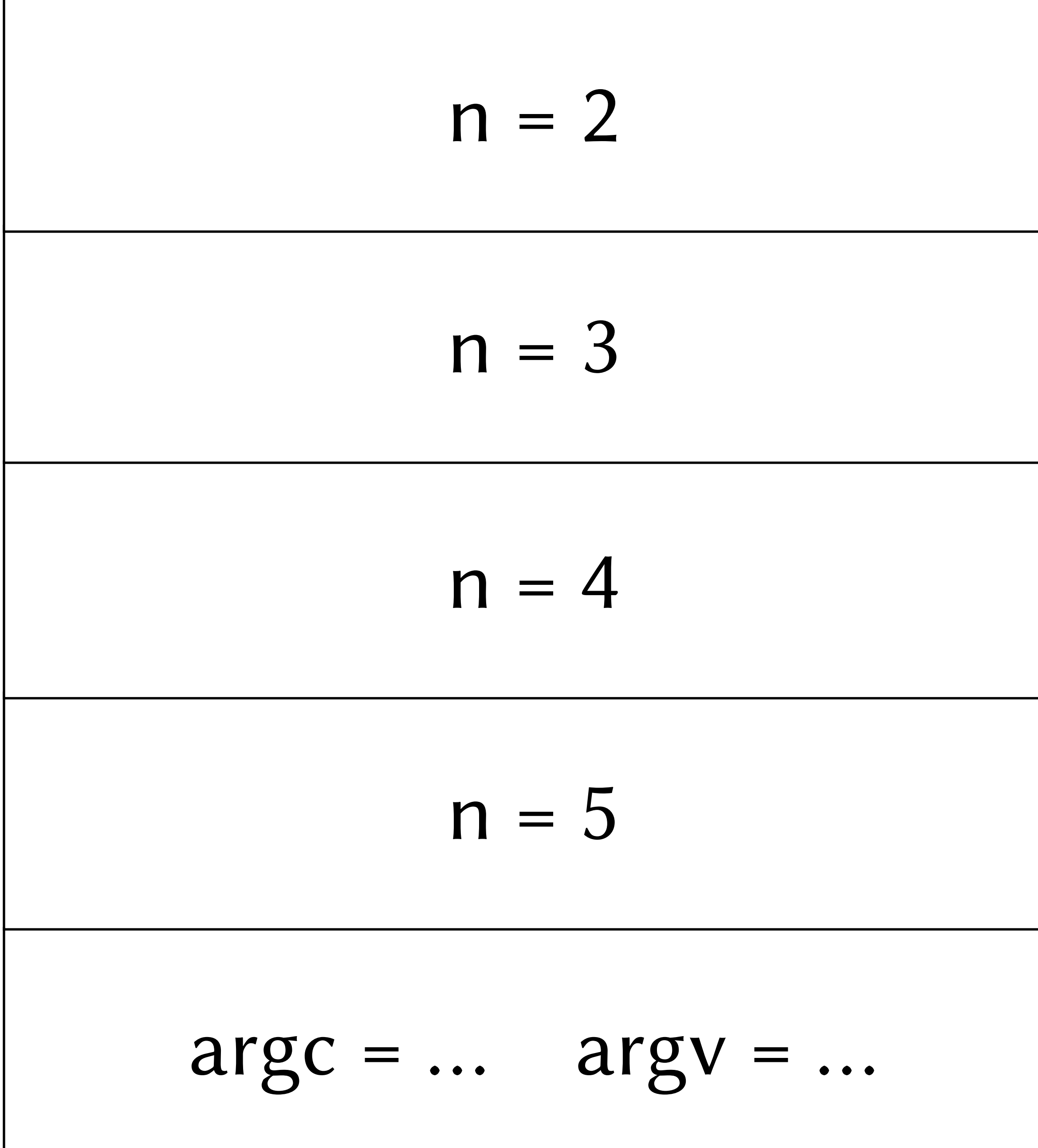
Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```



Stackanvändning i rekursiv fibonacci

```c
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```



Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 3
n = 4
n = 5
argc = … argv = …

Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 1
n = 3
n = 4
n = 5
argc = … argv = …

Stackanvändning i rekursiv fibonacci

```c
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 3
n = 4
n = 5
argc = …   argv = …

Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 4
n = 5
argc = …   argv = …

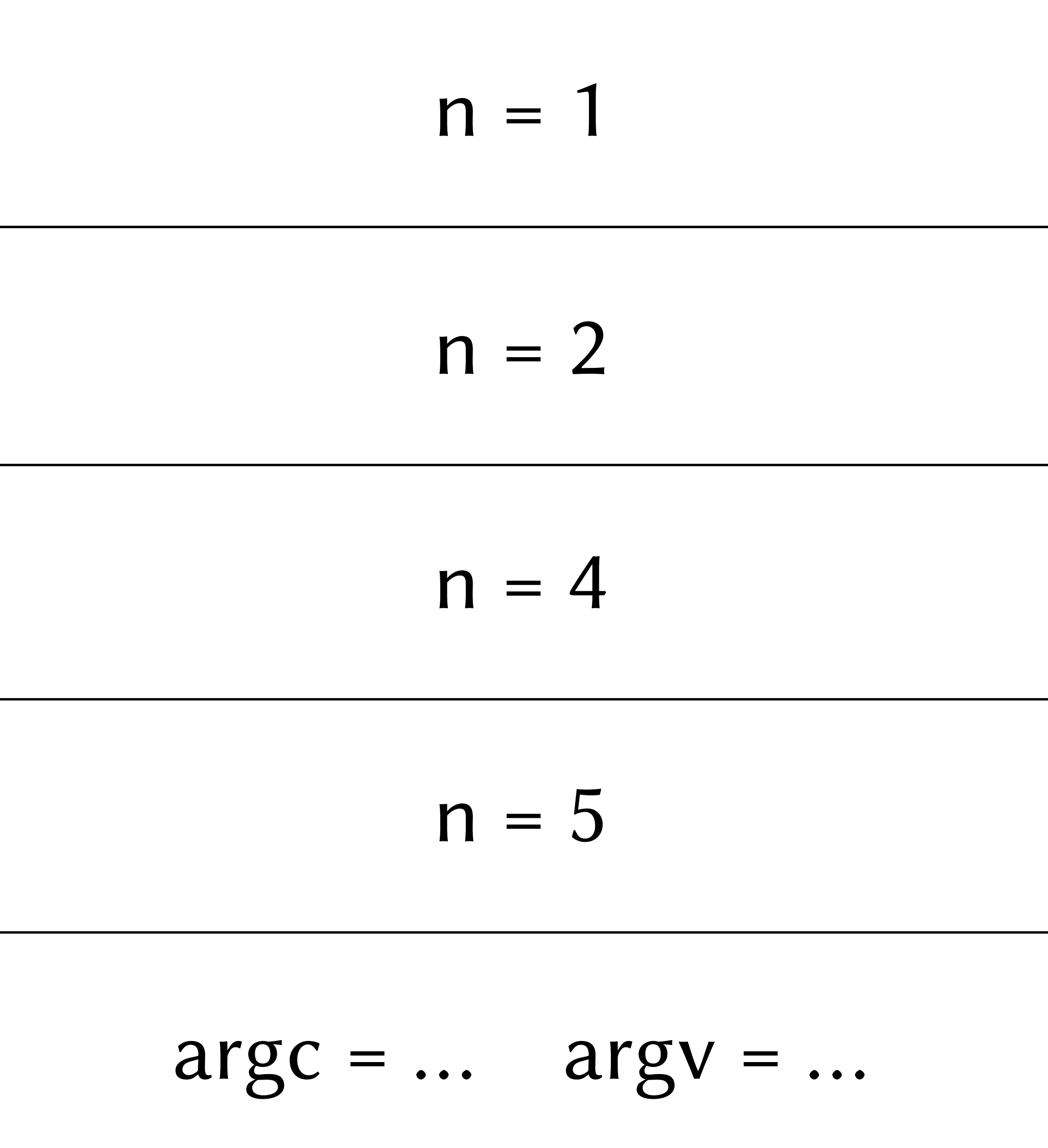
Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 2
n = 4
n = 5
argc = … argv = …

Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```



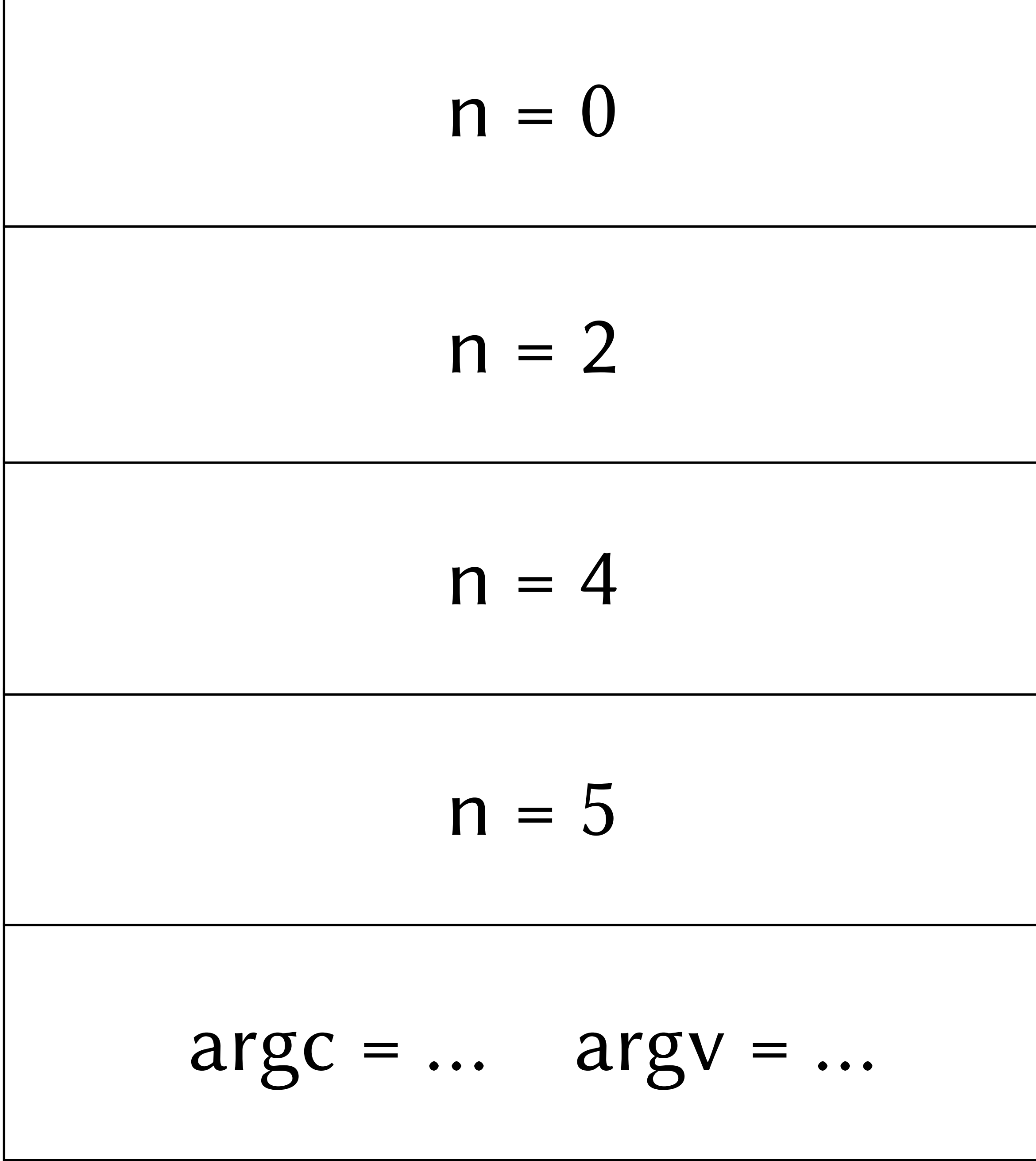
Stackanvändning i rekursiv fibonacci

```c
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 2
n = 4
n = 5
argc = … argv = …

Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```



Stackanvändning i rekursiv fibonacci

```c
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 2
n = 4
n = 5
argc = … argv = …

Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 4
n = 5
argc = …   argv = …

Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 5	
argc = …	argv = …

Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 3
n = 5
argc = … argv = …

Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 2
n = 3
n = 5
argc = … argv = …

Stackanvändning i rekursiv fibonacci

```c
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 1
n = 2
n = 3
n = 5
argc = … argv = …

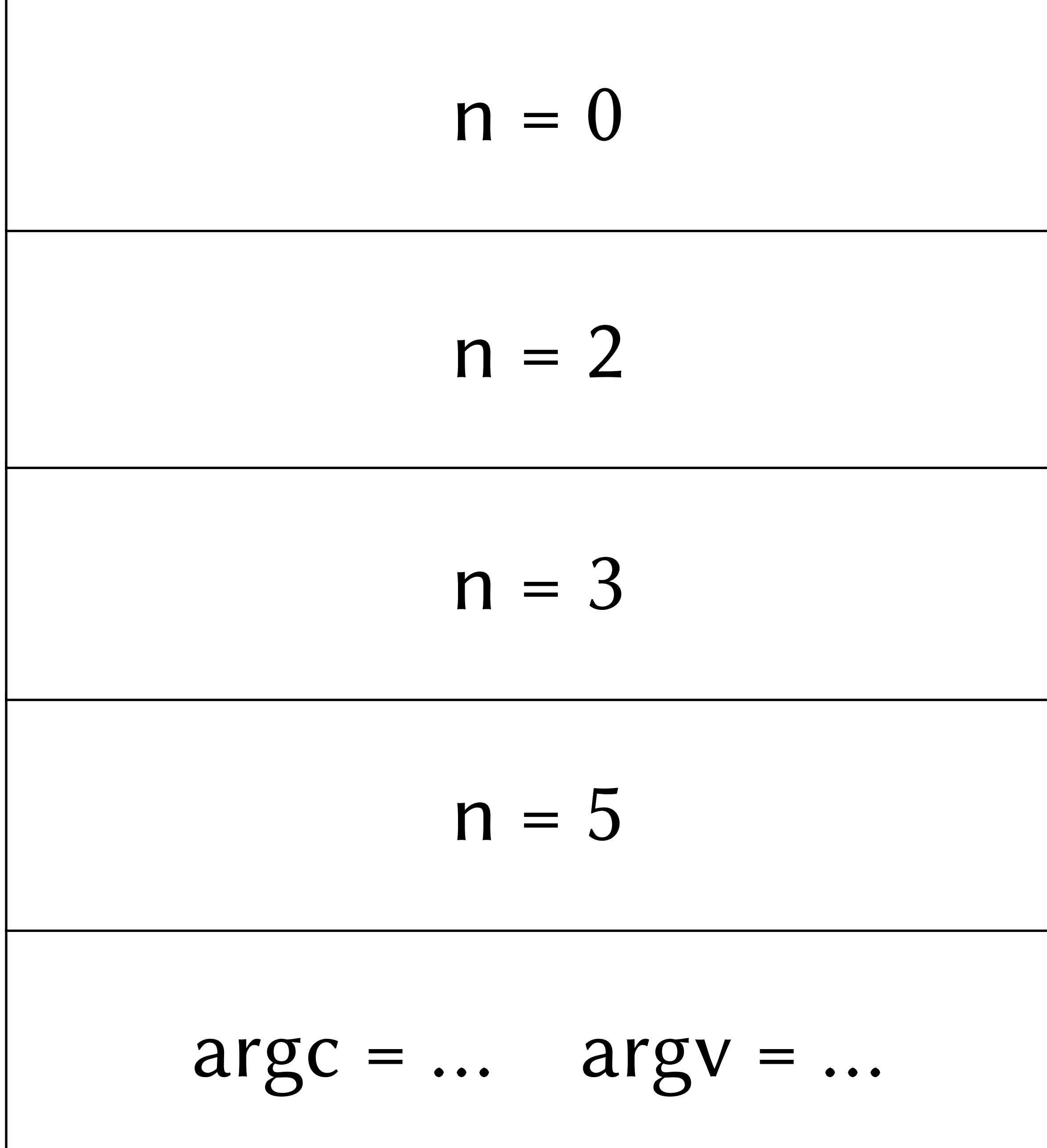
Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 2
n = 3
n = 5
argc = … argv = …

Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```



Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 2
n = 3
n = 5
argc = … argv = …

Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 3
n = 5
argc = … argv = …

Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 1
n = 3
n = 5
argc = … argv = …

Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

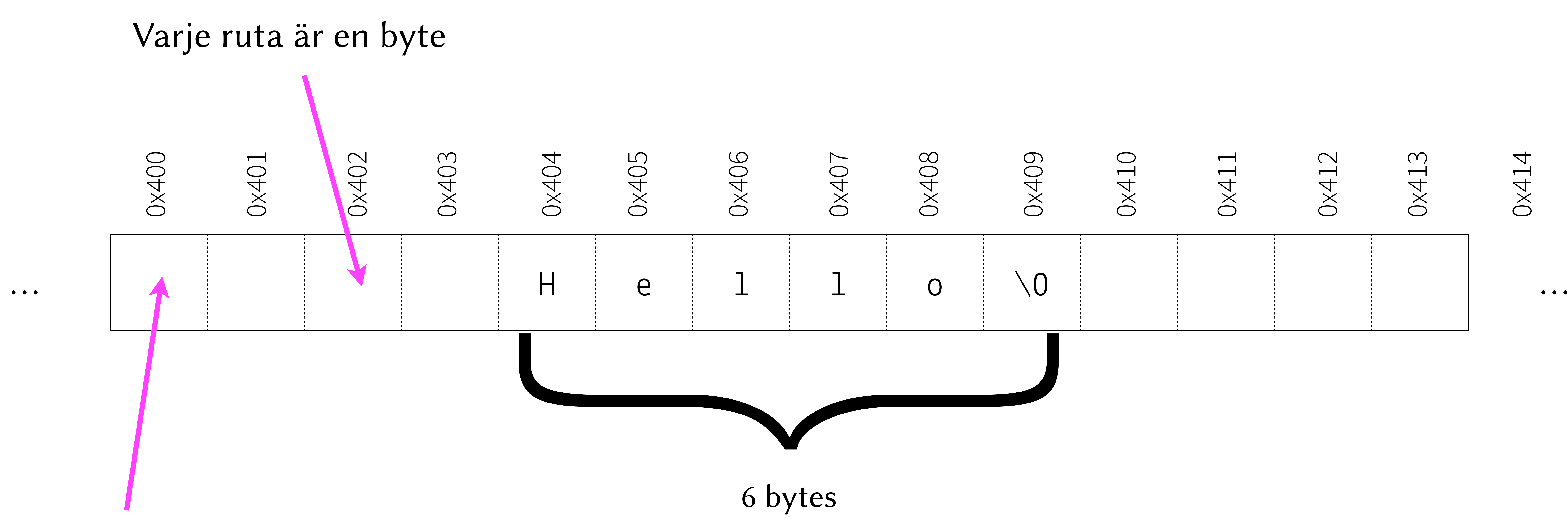
n = 3
n = 5
argc = …   argv = …

Stackanvändning i rekursiv fibonacci

```
int fib(int n) {
  return n == 0
    ? 0
    : n == 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

n = 5
argc = … argv = …

Minnet i C är som ett 1-dimensionellt rutat papper (Heapen)



Att rita på det rutade papperet

Steg 1: räkna ut hur många rutor vi behöver

Använd plattformsoberoende hjälpmedel och vanlig aritmetik

Steg 2: reservera motsvarande yta

Här kan det gå fel — det kanske inte finns plats på papperet

Steg 3: använd ytan hejvilt

Men sudda först!

Steg 4: lämna tillbaka platsen när du är klar

Annars kommer det gå dåligt i ett framtida steg 2

Att rita på det rutade papperet

Steg 1: räkna ut hur många rutor vi behöver

Använd plattformsoberoende hjälpmedel och vanlig aritmetik

sizeof(*T*) * *antal*

```
int size = sizeof(int) * 1024
```

Steg 2: reservera motsvarande yta

Här kan det gå fel — det kanske inte finns plats på papperet

T **namn* = malloc(*antal bytes*);

```
int *skonummer = malloc(size);
```

Steg 3: använd ytan hejvilt

Men sudda först!

(*Beror på datastrukturen*)

Steg 4: lämna tillbaka platsen när du är klar

Annars kommer det gå dåligt i ett framtida steg 2

free(*namn*);

```
free(skonummer);
```

Exempelprogram

```c
int main(int argc, char *argv[])
{
  int antal = atoi(argv[1]);
  int *skonummer = malloc(sizeof(int) * antal);
  for (int i = 0; i < antal; ++i) {
    skonummer[i] = 0; // nollställ arrayen
  }

  … // använd arrayen

  free(skonummer);

  return 0;
}
```

argc = … argv = … i = …
antal = … skonummer = …

?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Exempelprogram

```
int main(int argc, char *argv[])
{
  int antal = atoi(argv[1]);
  int *skonummer = malloc(sizeof(int) * antal);
  for (int i = 0; i < antal; ++i) {
    skonummer[i] = 0; // nollställ arrayen
  }

  … // använd arrayen

  free(skonummer);

  return 0;
}
```

argc = … argv = … i = …

antal = 8 skonummer =

?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Exempelprogram

```c
int main(int argc, char *argv[])
{
  int antal = atoi(argv[1]);
  int *skonummer = malloc(sizeof(int) * antal);
  for (int i = 0; i < antal; ++i) {
    skonummer[i] = 0; // nollställ arrayen
  }

  … // använd arrayen

  free(skonummer);

  return 0;
}
```

argc = … argv = … i = …
antal = 8 skonummer =

?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	0	0	0	0	0	0	0	0	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Dangling pointer

```c
int main(int argc, char *argv[])
{
  int antal = atoi(argv[1]);
  int *skonummer = malloc(sizeof(int) * antal);
  for (int i = 0; i < antal; ++i) {
    skonummer[i] = 0; // nollställ arrayen
  }

  … // använd arrayen

  free(skonummer);
  skonummer[2] = 42;

  return 0;
}
```

argc = … argv = … i = …
antal = 8 skonummer =

?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	0	0	0	0	0	0	0	0	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Minnesläckage!

```c
int main(int argc, char *argv[])
{
  int antal = atoi(argv[1]);
  int *skonummer = malloc(sizeof(int) * antal);
  for (int i = 0; i < antal; ++i) {
    skonummer[i] = 0; // nollställ arrayen
  }

  … // använd arrayen

  free(skonummer);
  skonummer = NULL;

  return 0;
}
```

argc = … argv = … i = …
antal = 8 skonummer = NULL

?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	0	0	0	0	0	0	0	0	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Rekommendation: både hängslen och livrem

```
int main(int argc, char *argv[])
{
  int antal = atoi(argv[1]);
  int *skonummer = malloc(sizeof(int) * antal);
  for (int i = 0; i < antal; ++i) {
    skonummer[i] = 0; // nollställ arrayen
  }

  … // använd arrayen

  free(skonummer);
  skonummer = NULL;

  return 0;
}
```

argc = … argv = … i = …
antal = 8 skonummer = NULL

?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Biblioteksfunktioner för att hantera heapminnne

- `sizeof(`T`)` — räkna ut hur stor typen T är i bytes

  Uträkningen sker vid kompilering, dvs. den är samma i alla körningar av programmet

Biblioteksfunktioner för att hantera heapminnne

- `sizeof`(T) — räkna ut hur stor typen T är i bytes

  Uträkningen sker vid kompilering, dvs. den är samma i alla körningar av programmet

- `malloc`(s) — allokera s ”osuddade” bytes på heapen

  Returnerar adressen (dvs. en pekare) till de allokerade utrymmet

Biblioteksfunktioner för att hantera heapminnne

- `sizeof(`T`)` — räkna ut hur stor typen T är i bytes

  Uträkningen sker vid kompilering, dvs. den är samma i alla körningar av programmet

- `malloc(`s`)` — allokera s ”osuddade” bytes på heapen

  Returnerar adressen (dvs. en pekare) till de allokerade utrymmet

- `calloc(`n`, `s`)` — allokera $n \times s$ suddade bytes på heapen

  Returnerar adressen (dvs. en pekare) till de allokerade utrymmet

Biblioteksfunktioner för att hantera heapminnne

- `sizeof(`T`)` — räkna ut hur stor typen T är i bytes

  Uträkningen sker vid kompilering, dvs. den är samma i alla körningar av programmet

- `malloc(`s`)` — allokera s ”osuddade” bytes på heapen

  Returnerar adressen (dvs. en pekare) till de allokerade utrymmet

- `calloc(`n`, `s`)` — allokera $n \times s$ suddade bytes på heapen

  Returnerar adressen (dvs. en pekare) till de allokerade utrymmet

- `realloc(`a`, `s`)` — ändra storleken på allokeringen a till s bytes

  a skickas in som en pekare, returnerar a alternativt en ny adress om utrymmet behövde flyttas för att kunna rymma s bytes

Biblioteksfunktioner för att hantera heapminnne

- `sizeof(`T`)` — räkna ut hur stor typen T är i bytes

  Uträkningen sker vid kompilering, dvs. den är samma i alla körningar av programmet

- `malloc(`s`)` — allokera s ”osuddade” bytes på heapen

  Returnerar adressen (dvs. en pekare) till de allokerade utrymmet

- `calloc(`n`, `s`)` — allokera $n \times s$ suddade bytes på heapen

  Returnerar adressen (dvs. en pekare) till de allokerade utrymmet

- `realloc(`a`, `s`)` — ändra storleken på allokeringen a till s bytes

  a skickas in som en pekare, returnerar a alternativt en ny adress om utrymmet behövde flyttas för att kunna rymma s bytes

- `free(`a`)` — gör allokeringen a tillgänglig för framtida anrop till `malloc` och `calloc`

Minnesläckage

Ett program som allokerar minne men som glömmer att lämna tillbaka det ”läcker minne”

Tumregel:

Varje anrop till `malloc` skall paras ihop med ett motsvarande `free`-anrop som lämnar tillbaka minnet

Skriv in dem i programmet i par

Kan vara mycket svårt att avgöra *var* och *när* `free` skall göras

Verktyg för att detektera minnesläckage tas upp på kursen

Exemplifieras med valgrind, men många andra finns!

Krav på avsaknad av minnesläckage i mål Z100 och Z101

Pekare och pekarvariabler

En pekare är en adress till en ”ruta på papperet”

Om vi ser minnet som en 1-dimensionell array av bytes är en pekare ett index in i arrayen

Pekare och pekarvariabler

En pekare är en adress till en ”ruta på papperet”

Om vi ser minnet som en 1-dimensionell array av bytes är en pekare ett index in i arrayen

Alla värden i ett program ligger någonstans i minnet, dvs. det har en adress

Vi kan få adressen med hjälp av adresstagningsoperatorn &

Pekare och pekarvariabler

En pekare är en adress till en ”ruta på papperet”

Om vi ser minnet som en 1-dimensionell array av bytes är en pekare ett index in i arrayen

Alla värden i ett program ligger någonstans i minnet, dvs. det har en adress

Vi kan få adressen med hjälp av adresstagningsoperatorn &

Pekarvariabler (*T* * — en pekare till en *T*) lagrar adresser till platser i minnet

För att följa en pekare till den plats i minnet dit den pekar ***avrefererar*** vi den med operatorn *

Pekare och pekarvariabler

En pekare är en adress till en ”ruta på papperet”

Om vi ser minnet som en 1-dimensionell array av bytes är en pekare ett index in i arrayen

Alla värden i ett program ligger någonstans i minnet, dvs. det har en adress

Vi kan få adressen med hjälp av adresstagningsoperatorn &

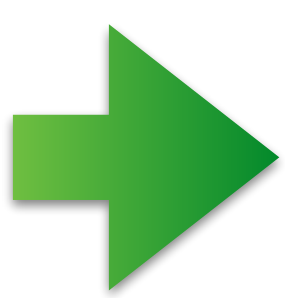
Pekarvariabler (*T* * — en pekare till en *T*) lagrar adresser till platser i minnet

För att följa en pekare till den plats i minnet dit den pekar ***avrefererar*** vi den med operatorn *

En tilldelning till en pekarvariabel påverkar inte det minne som den pekar ut

Undantag: en variabel som pekar på ”sitt eget minne”

Exempelprogram

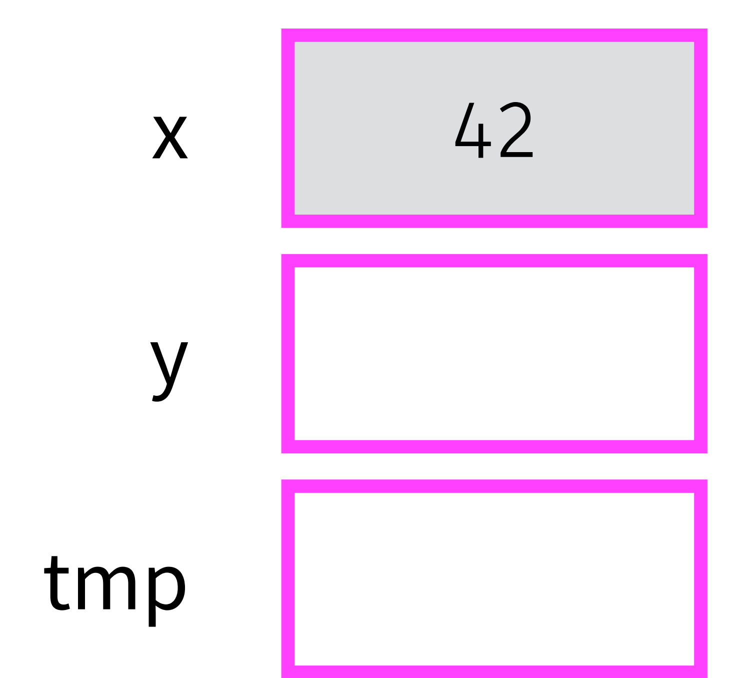


```
int x = 42;      // x innehåller ett heltal
int y = 4711;

int *tmp = &x;

tmp = &y;

*tmp = 12;
```

x	???
y	???
tmp	???

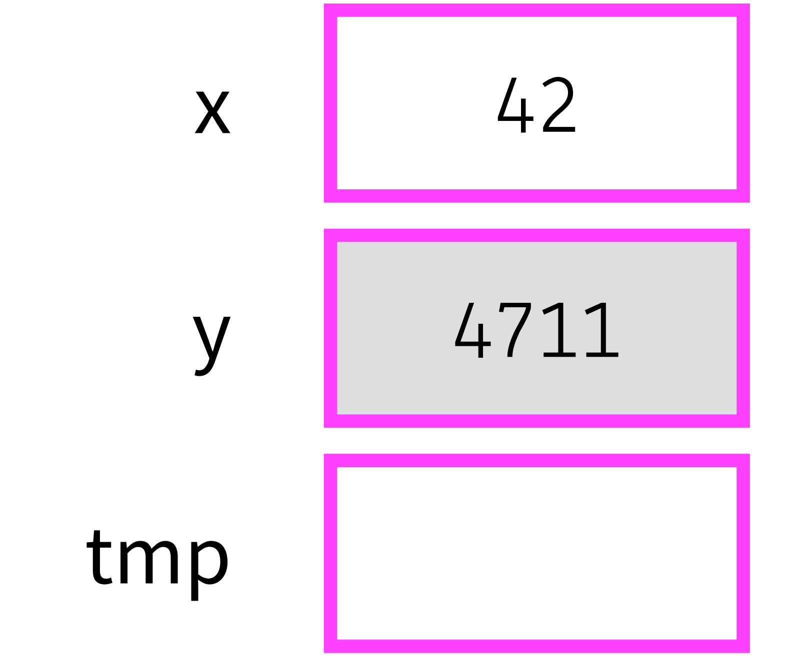
Exempelprogram

```
int x = 42;
int y = 4711;   // y innehåller ett heltal

int *tmp = &x;

tmp = &y;

*tmp = 12;
```



Exempelprogram

```
int x = 42;
int y = 4711;

int *tmp = &x;

tmp = &y;

*tmp = 12;
```

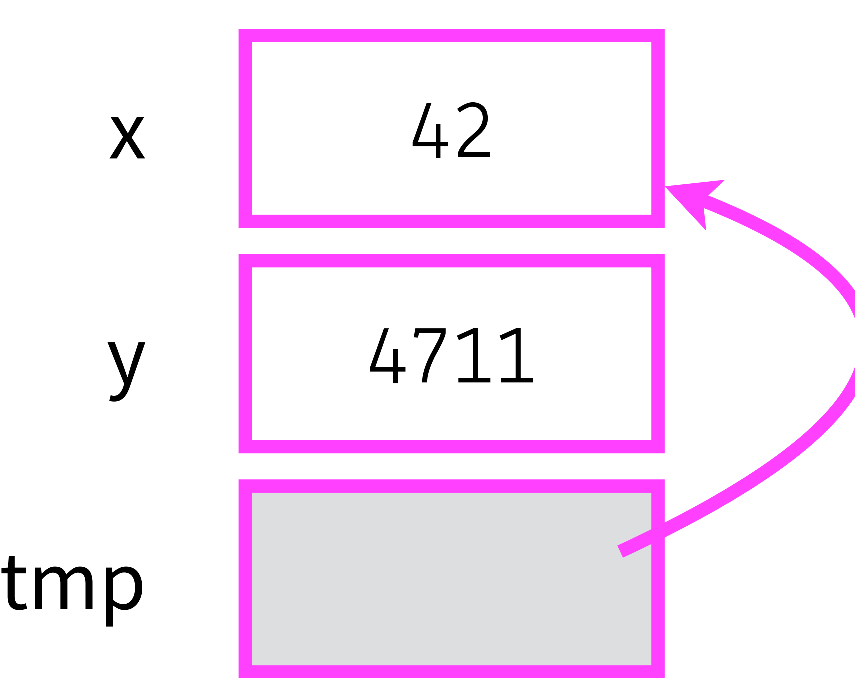


Exempelprogram

adresstagnings-
operatorn

&x

```
int x = 42;
int y = 4711;

int *tmp = &x;  // tmp innehåller adressen
                // till en plats i minnet där
tmp = &y;       // x's heltal är lagrat

*tmp = 12;
```



Exempelprogram

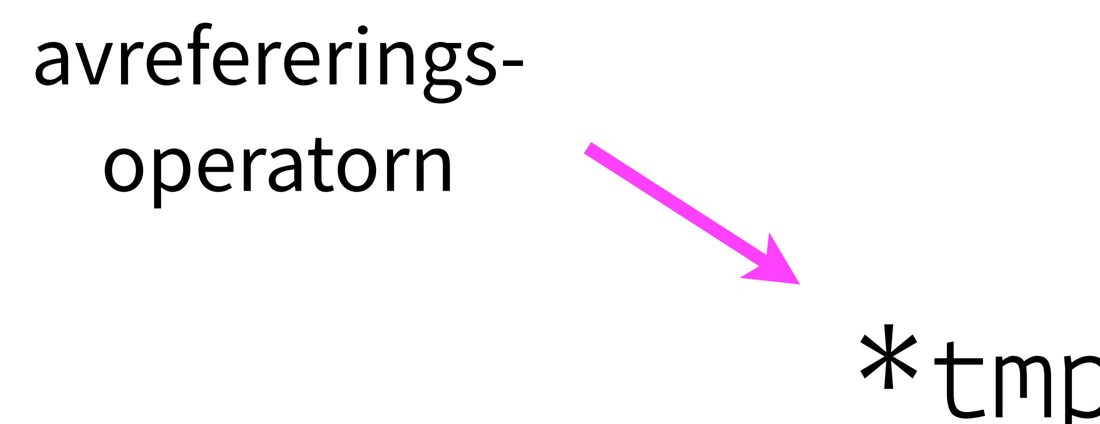
```
int x = 42;
int y = 4711;

int *tmp = &x;

tmp = &y;          // tmp innehåller adressen
                   // till en plats i minnet där
*tmp = 12;         // y’s heltal är lagrat
```

x 42

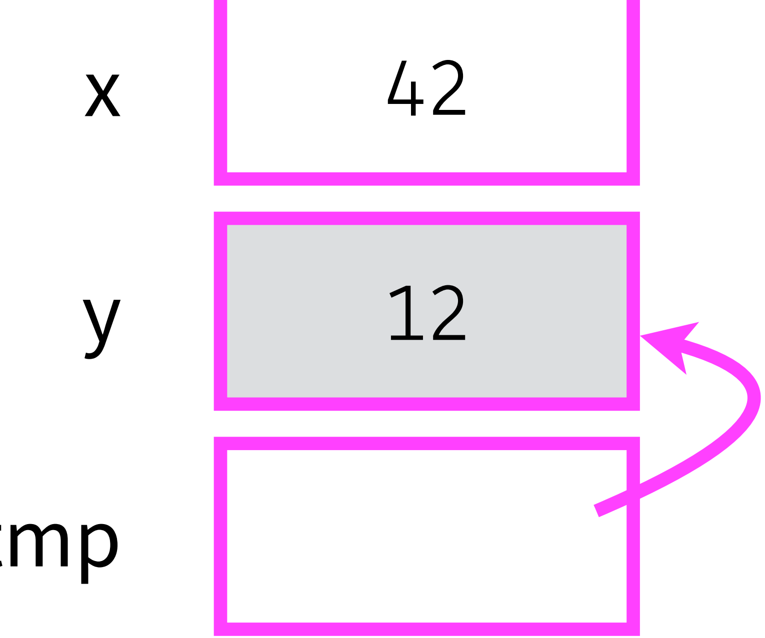
y 4711

tmp

Exempelprogram

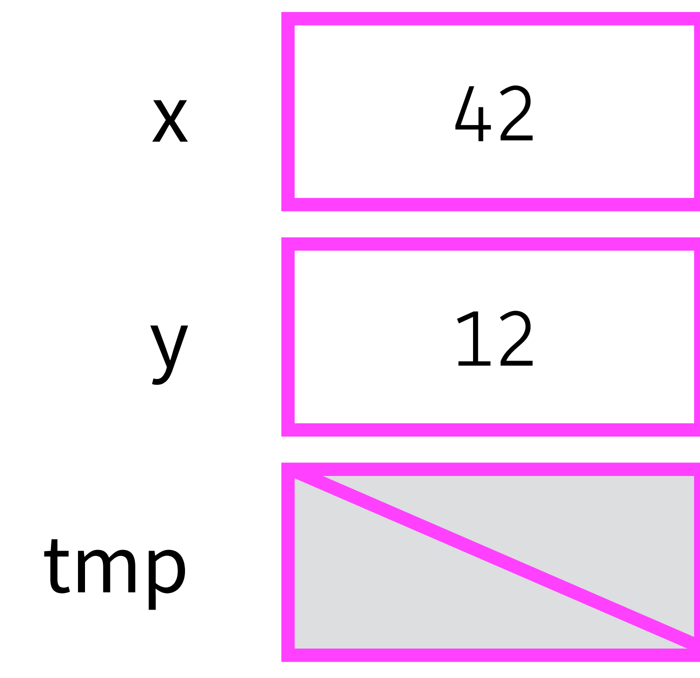



```
int x = 42;
int y = 4711;

int *tmp = &x;

tmp = &y;

*tmp = 12;        // uppdatera minnesplatsen som
                  // tmp pekar på
```



Exempelprogram

```
int x = 42;
int y = 4711;

int *tmp = &x;

tmp = &y;

*tmp = 12;

tmp = NULL;
```

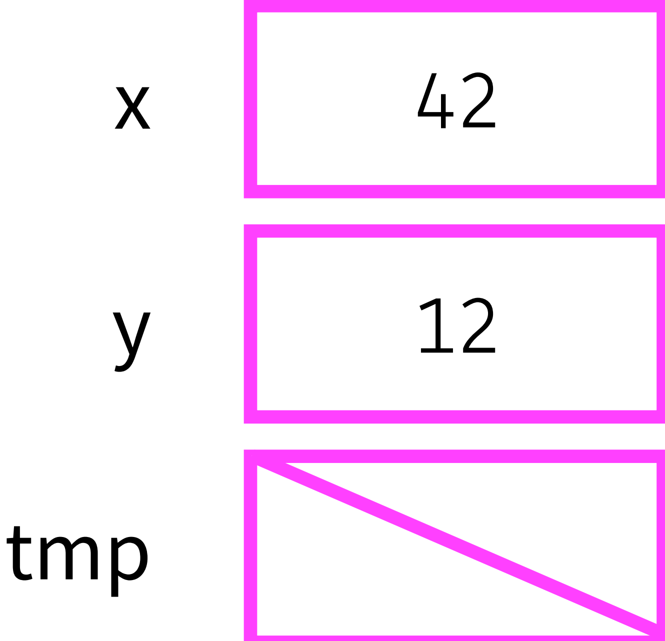


”I call it my billion-dollar mistake. It was the invention of the null reference in 1965”

— *Tony Hoare*

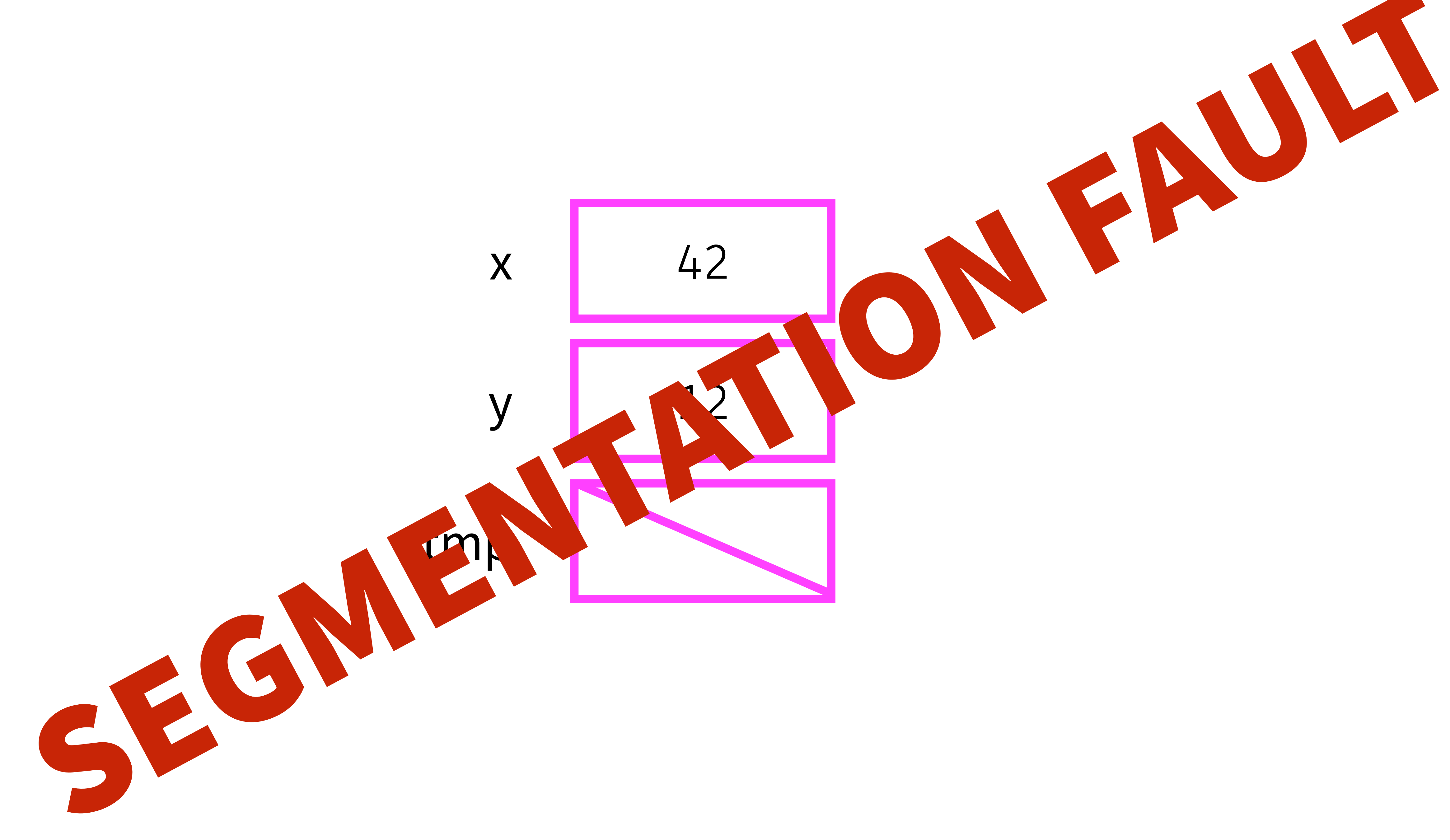
Exempelprogram

```
int x = 42;
int y = 4711;

int *tmp = &x;

tmp = &y;

*tmp = 12;

tmp = NULL;

*tmp = 12;
```



Exempelprogram

```
int x = 42;
int y = 4711;

int *tmp = &x;

tmp = &y;

*tmp = 12;

tmp = NULL;

*tmp = 12;
```



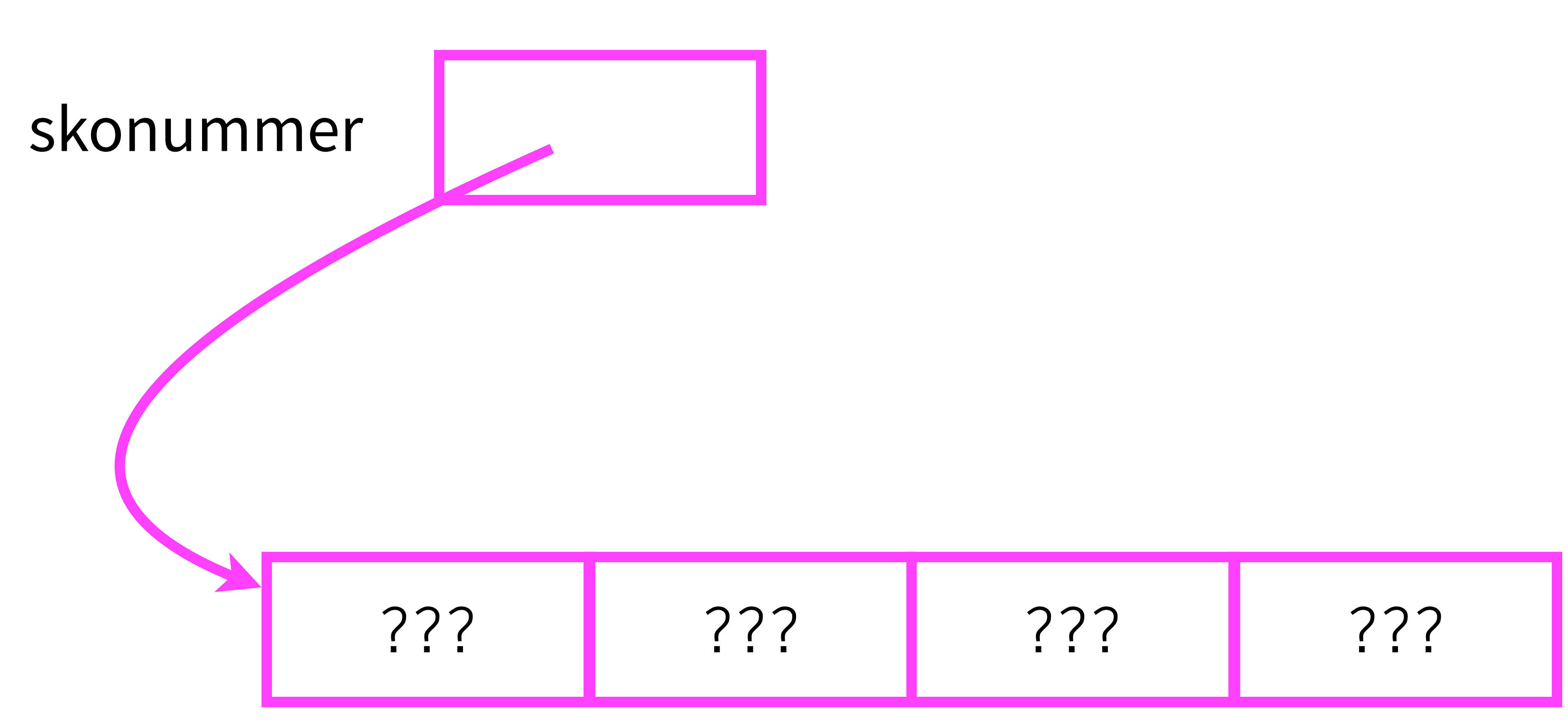
Exempelprogram

```
int *skonummer;

skonummer = malloc(4 * sizeof(int));

skonummer[2] = 44;

free(skonummer);

skonummer[2] = 38;
```

skonummer ???

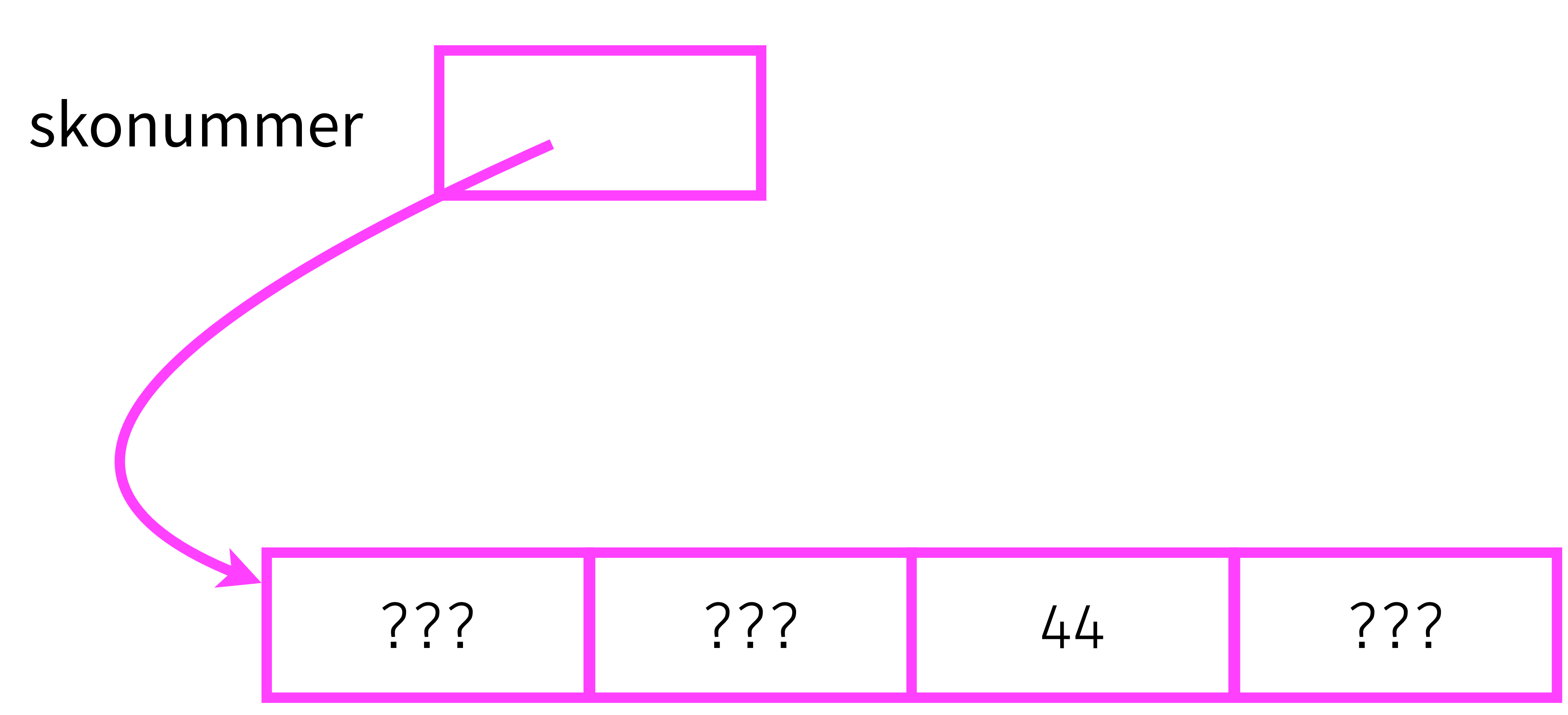
Exempelprogram

```
int *skonummer;
skonummer = malloc(4 * sizeof(int));
skonummer[2] = 44;
free(skonummer);
skonummer[2] = 38;
```



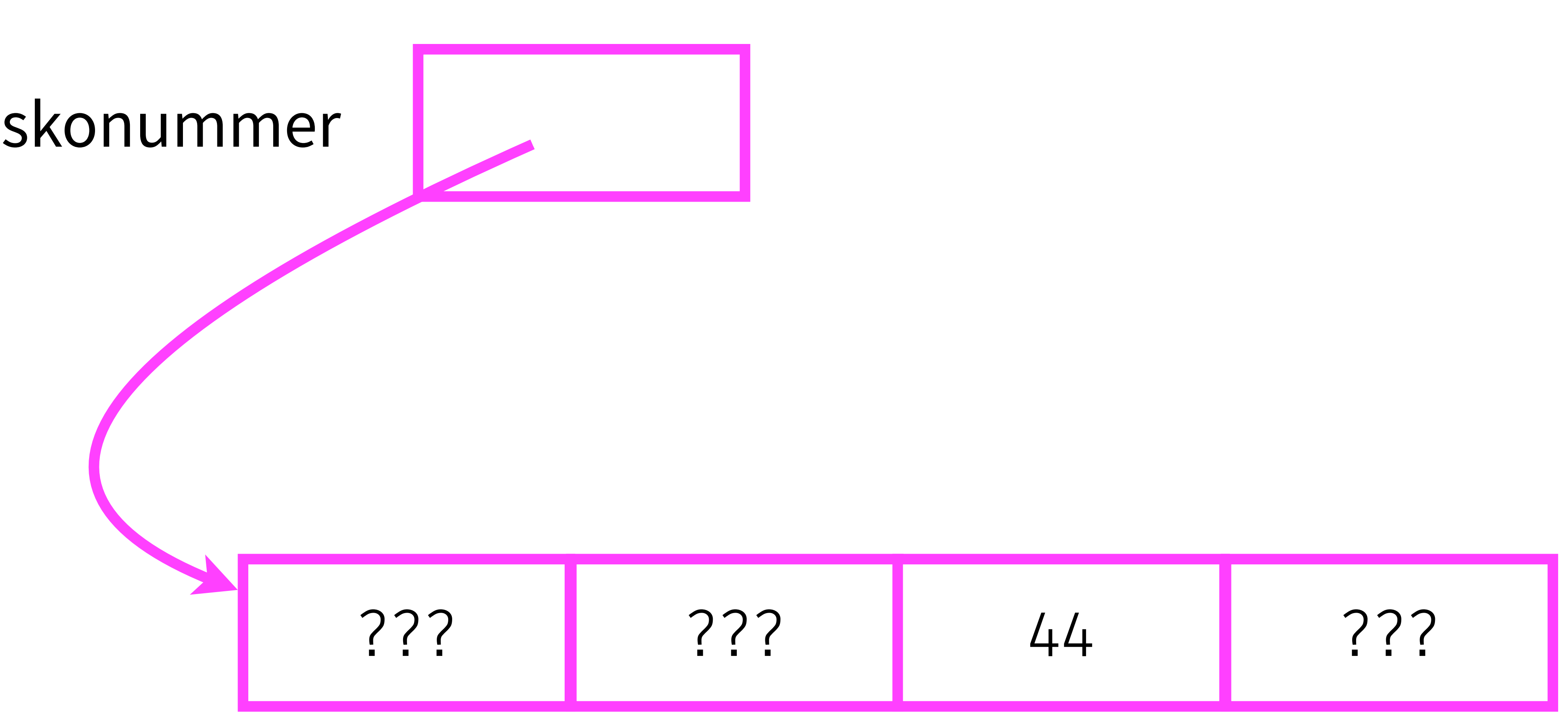
Exempelprogram

```
int *skonummer;

skonummer = malloc(4 * sizeof(int));

skonummer[2] = 44;

free(skonummer);

skonummer[2] = 38;
```



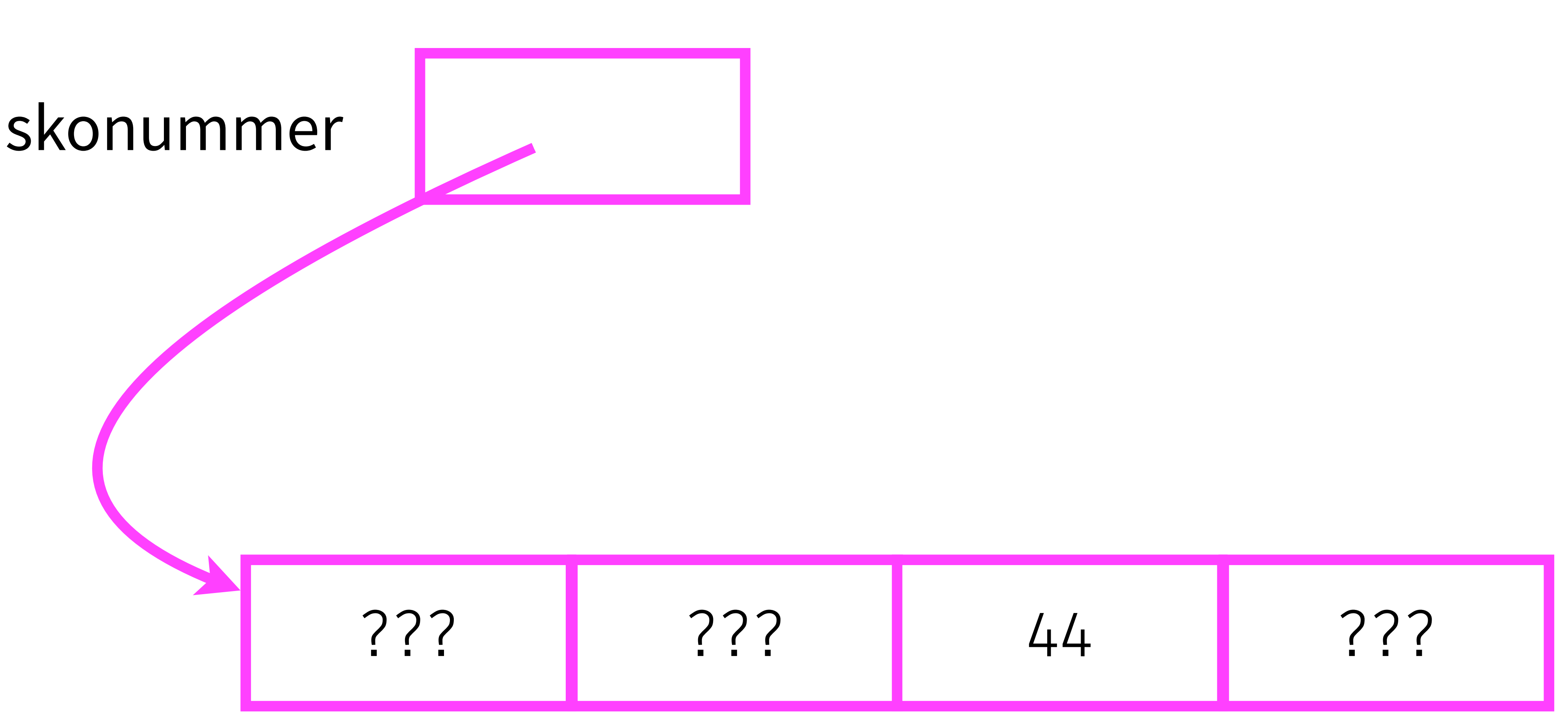
Exempelprogram

```
int *skonummer;

skonummer = malloc(4 * sizeof(int));

skonummer[2] = 44;

free(skonummer);

skonummer[2] = 38;
```



Exempelprogram

```
int *skonummer;

skonummer = malloc(4 * sizeof(int));

skonummer[2] = 44;

free(skonummer);

skonummer[2] = 38;
```



Exempelprogram

```
int *skonummer;

skonummer = malloc(4 * sizeof(int));

skonummer[2] = 44;

free(skonummer);

skonummer[2] = 38;
```

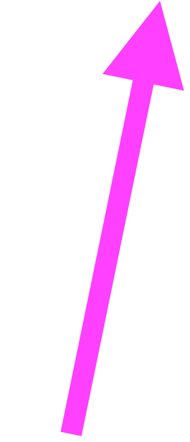
skonummer

???	???	44	???

UNDEFINED BEHAVIOUR

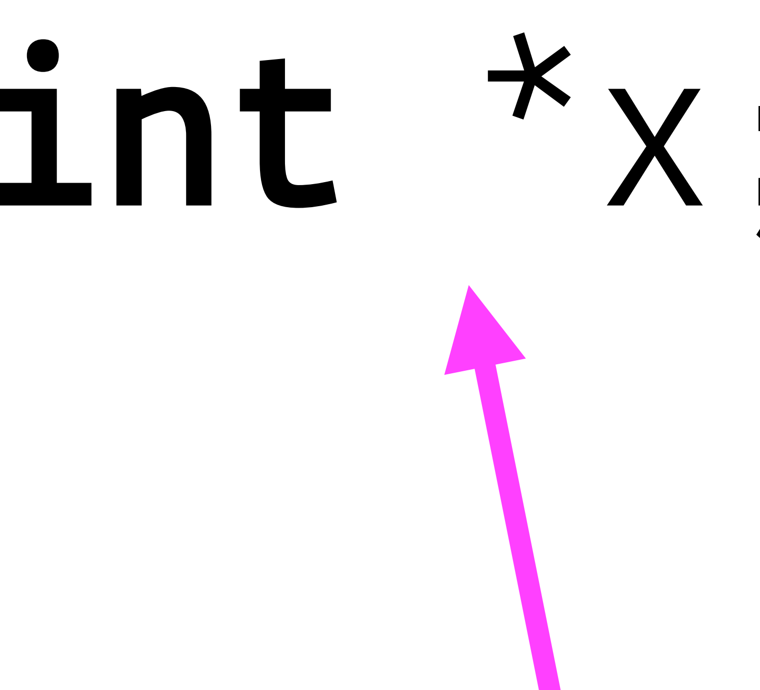
Pekartypen

```
int x;
```



x är en variabel som innehåller ett heltal

```
int *x;
```



x är en variabel som innehåller en pekare till en plats i minnet där det finns (åtminstone) ett heltal

Ny typ: void *

En pekare till något av okänd typ

C tillåter **inte** att den avrefereras

Användbar för att skapa t.ex. generella lagringsklasser (se Inlämningsuppgift 1)

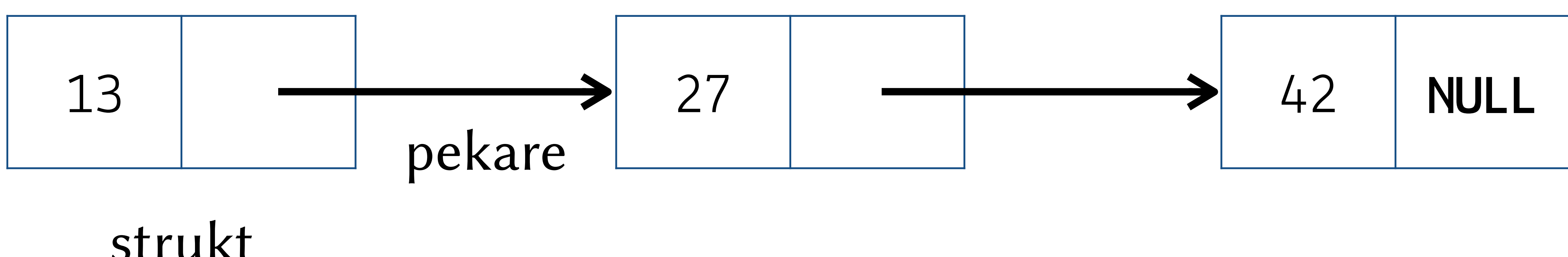
```c
int x;
int *y = &x;
void *z = y;

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

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

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

Tas upp mer senare

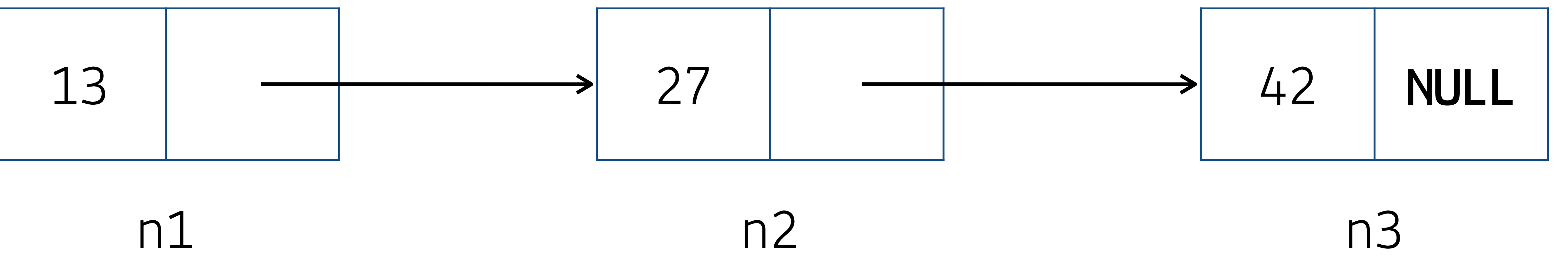
Länkade datastrukturer

- Hittills har vi endast sett strukturer vars storlek måste vara känd ”ahead-of-time"
  - En strukt består av ett fixt antal poster, alla med fix storlek
  - En array består av ett fixt antal element, med samma, fixa storlek
- Hur skall vi hantera fallet då vi inte vet hur många av ett visst data vi vill ha?
  - Hur långt är ett namn?
  - Hur många studenter på kursen?
  - ...
- Detta kan vi lösa med hjälp av pekare!

Ritnotation:



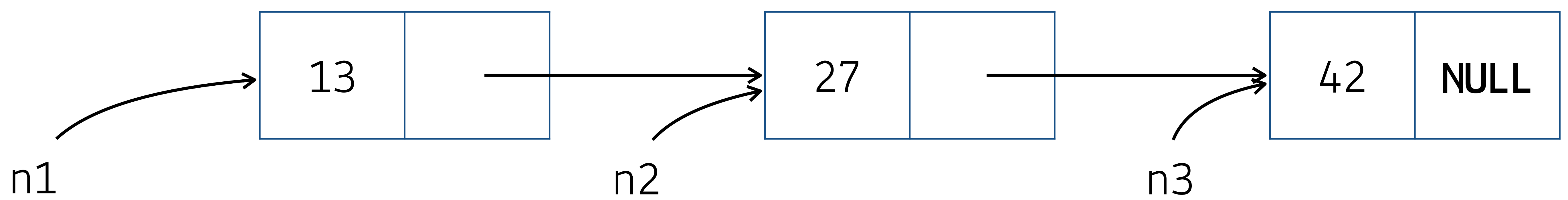
Pekare och länkade strukturer [På stacken]




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

```
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 };
```

Pekare och länkade strukturer [På heapen]




```c
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 };
```

Avreferering av pekare i länkade strukturer

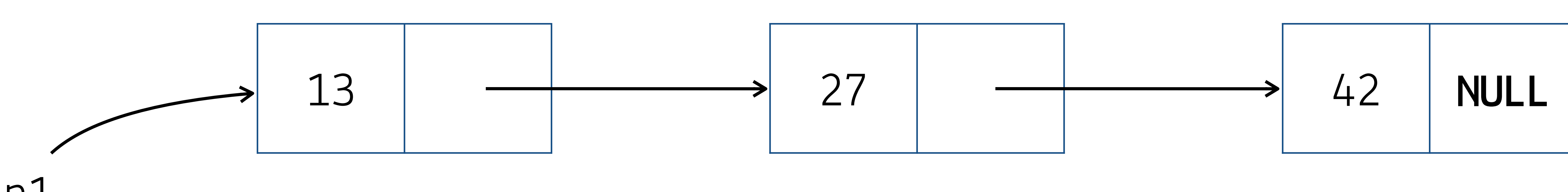
- För att komma till 27

```
(*(*n1).next).number
```

- För att komma till 42

```
(*(*(*n1).next).next).number
```

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



Avreferering av pekare i länkade strukturer

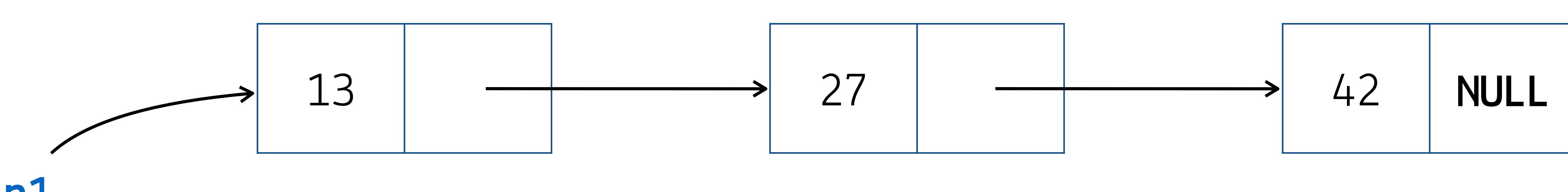
- För att komma till 27

```
(*(*n1).next).number
```

- För att komma till 42

```
(*(*(*n1).next).next).number
```

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



Avreferering av pekare i länkade strukturer

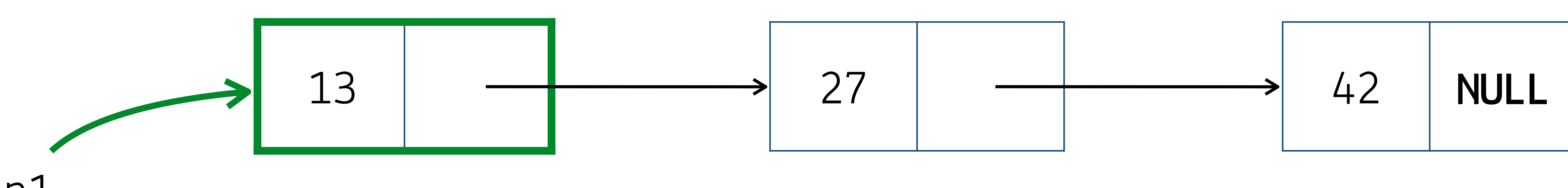
- För att komma till 27

```
(*(*n1).next).number
```

- För att komma till 42

```
(*(*(*n1).next).next).number
```

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



Avreferering av pekare i länkade strukturer

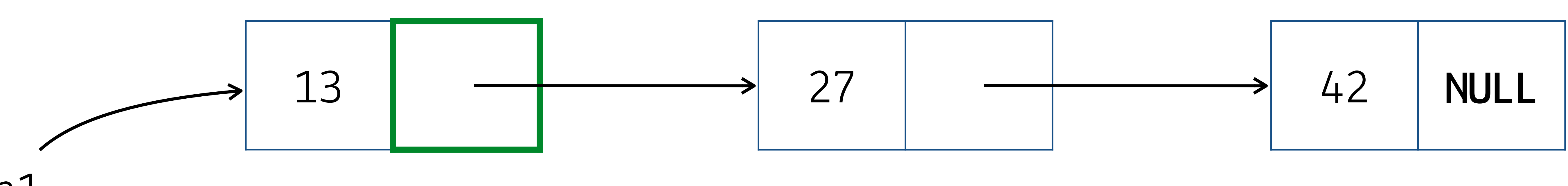
- För att komma till 27

```
(*(*n1).next).number
```

- För att komma till 42

```
(*(*(*n1).next).next).number
```

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



Avreferering av pekare i länkade strukturer

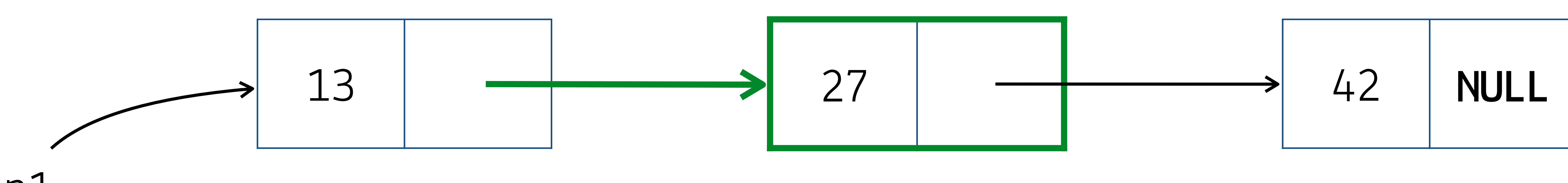
- För att komma till 27

  `(*(*n1).next).number`

- För att komma till 42

  `(*(*(*n1).next).next).number`

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



Avreferering av pekare i länkade strukturer

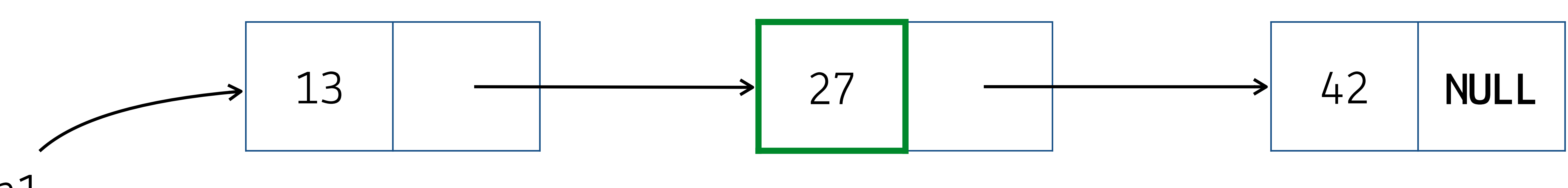
- För att komma till 27

  ```
  (*(*n1).next).number
  ```

- För att komma till 42

  ```
  (*(*(*n1).next).next).number
  ```

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



Avreferering av pekare i länkade strukturer

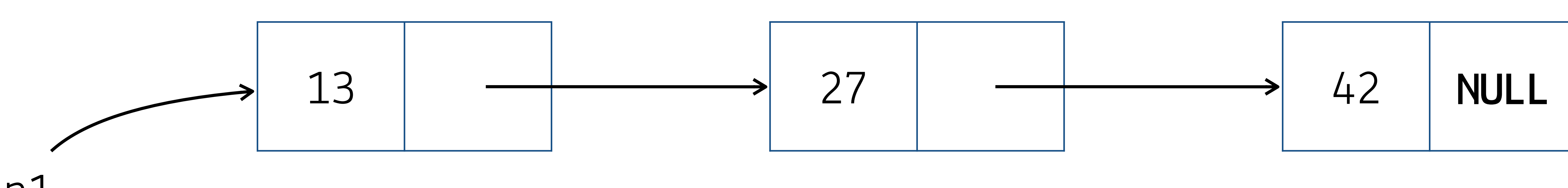
- För att komma till 27

```
(*(*n1).next).number
```

- För att komma till 42

```
(*(*(*n1).next).next).number
```

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



Avreferering av pekare i länkade strukturer

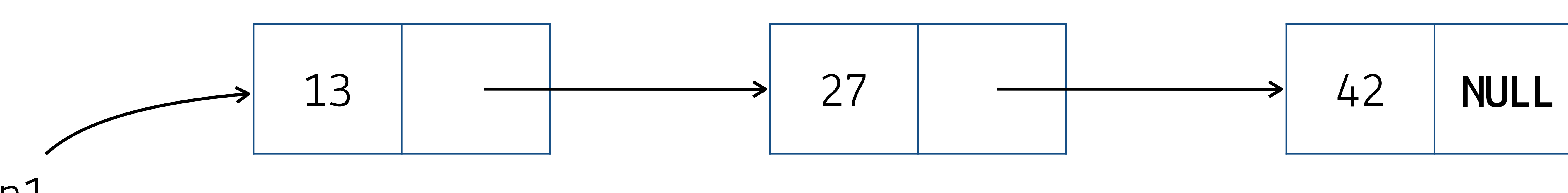
- För att komma till 27

```
n1->next->number
```

- För att komma till 42

```
n1->next->next->number
```

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



Avreferering av pekare i länkade strukturer

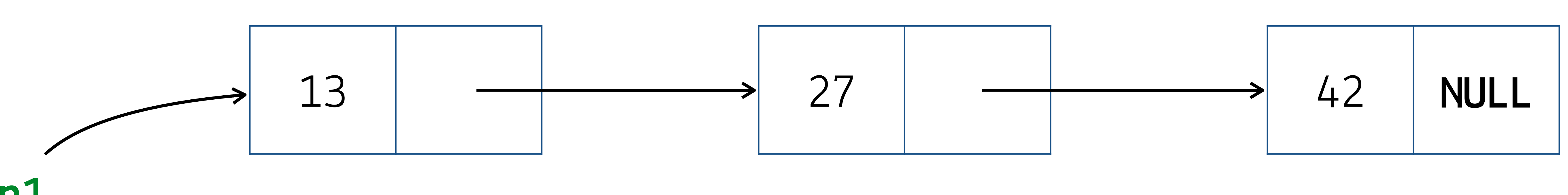
- För att komma till 27

  `n1->next->number`

- För att komma till 42

  `n1->next->next->number`

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



Avreferering av pekare i länkade strukturer

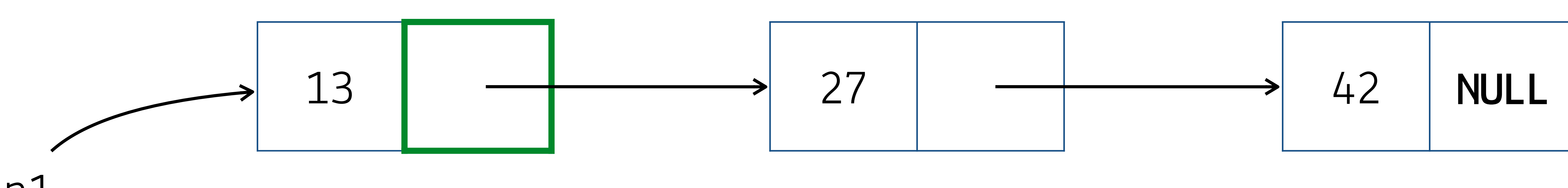
- För att komma till 27

  `n1->next``->number`

- För att komma till 42

  `n1->next->next->number`

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



Avreferering av pekare i länkade strukturer

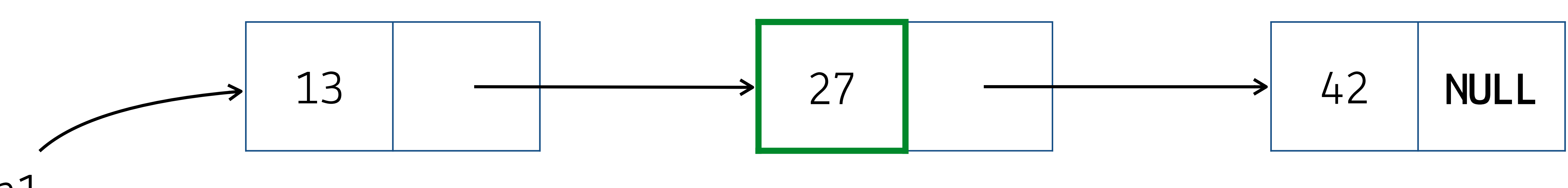
- För att komma till 27

  `n1->next->number`

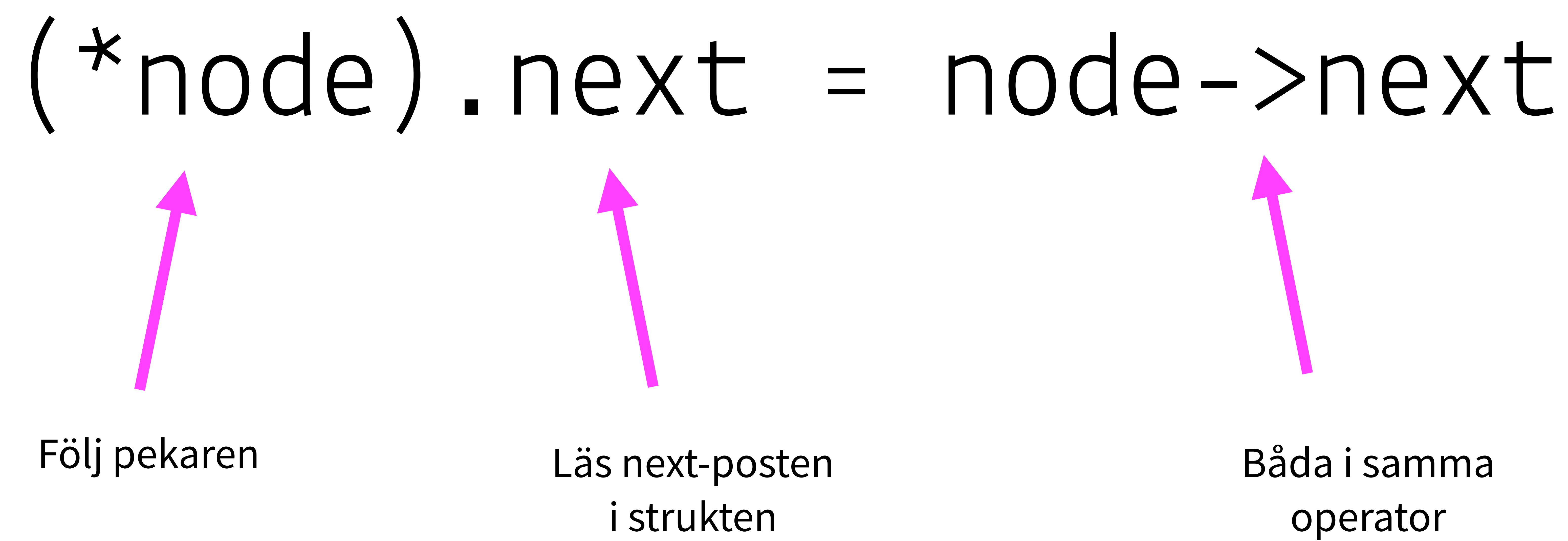
- För att komma till 42

  `n1->next->next->number`

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



Tydligare syntax för att navigera länkade strukturer



Minneshantering

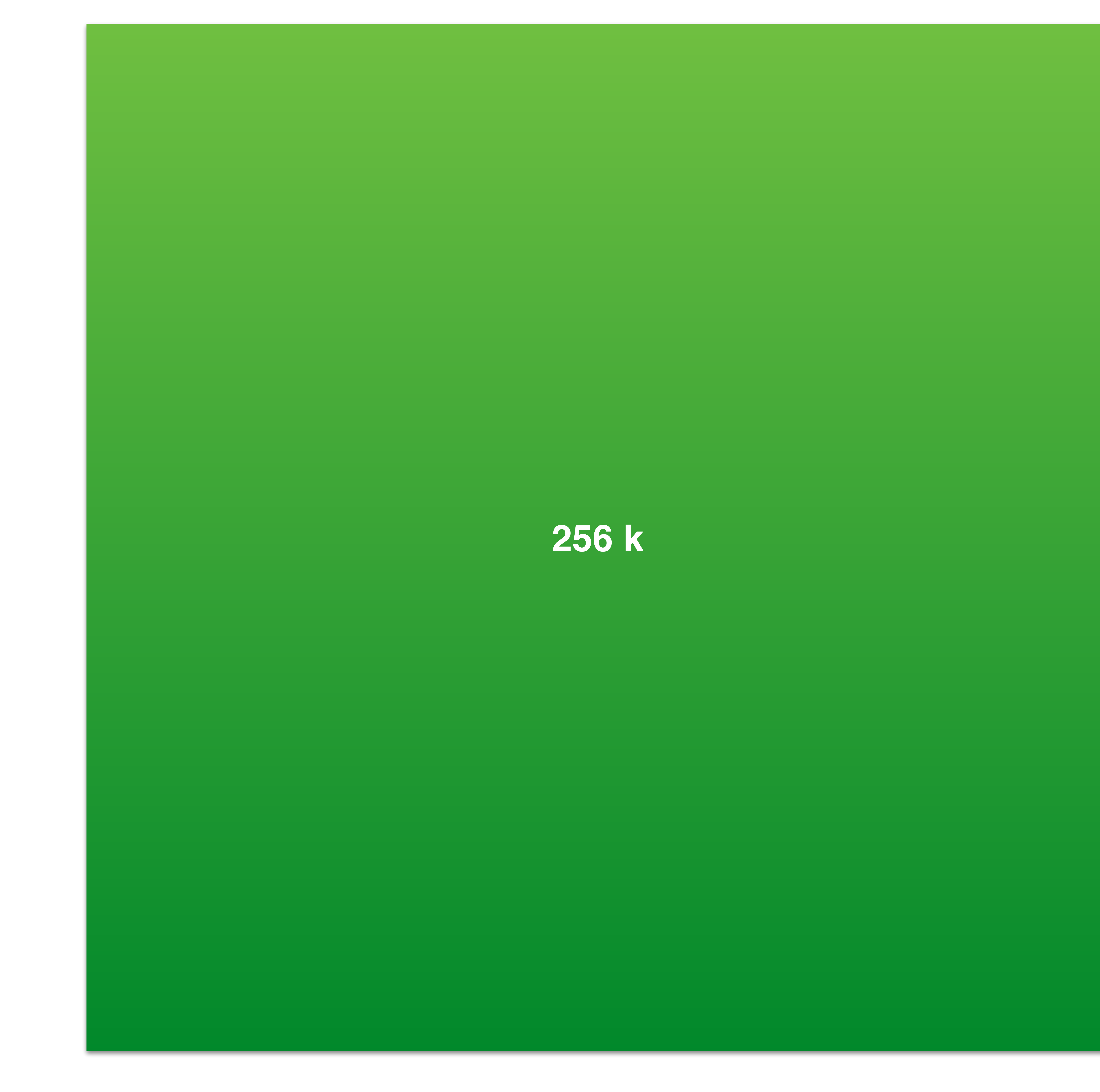
Allokera minne explicit med ngn rutin (t.ex. `malloc`)

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?

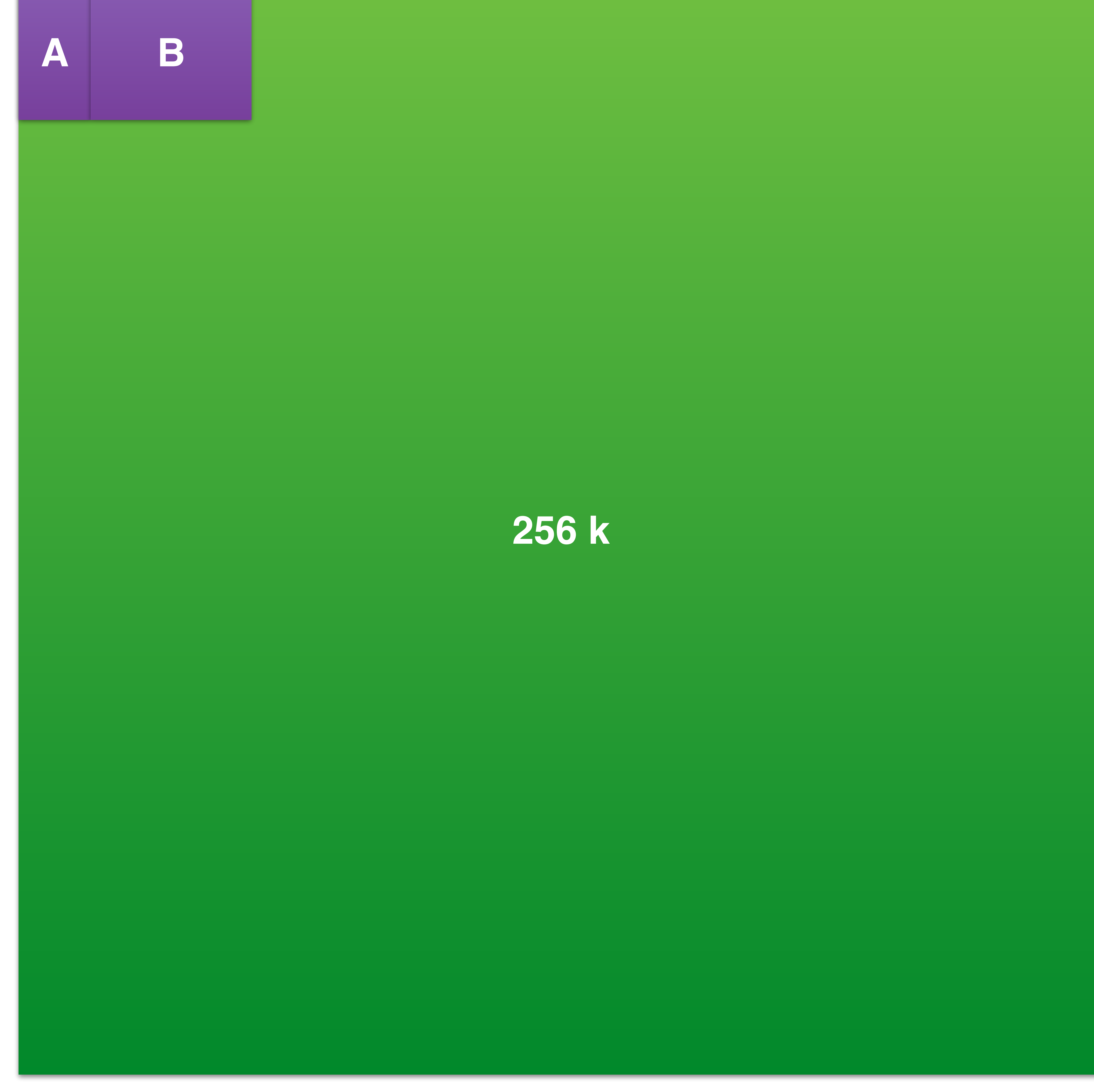

256 k

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

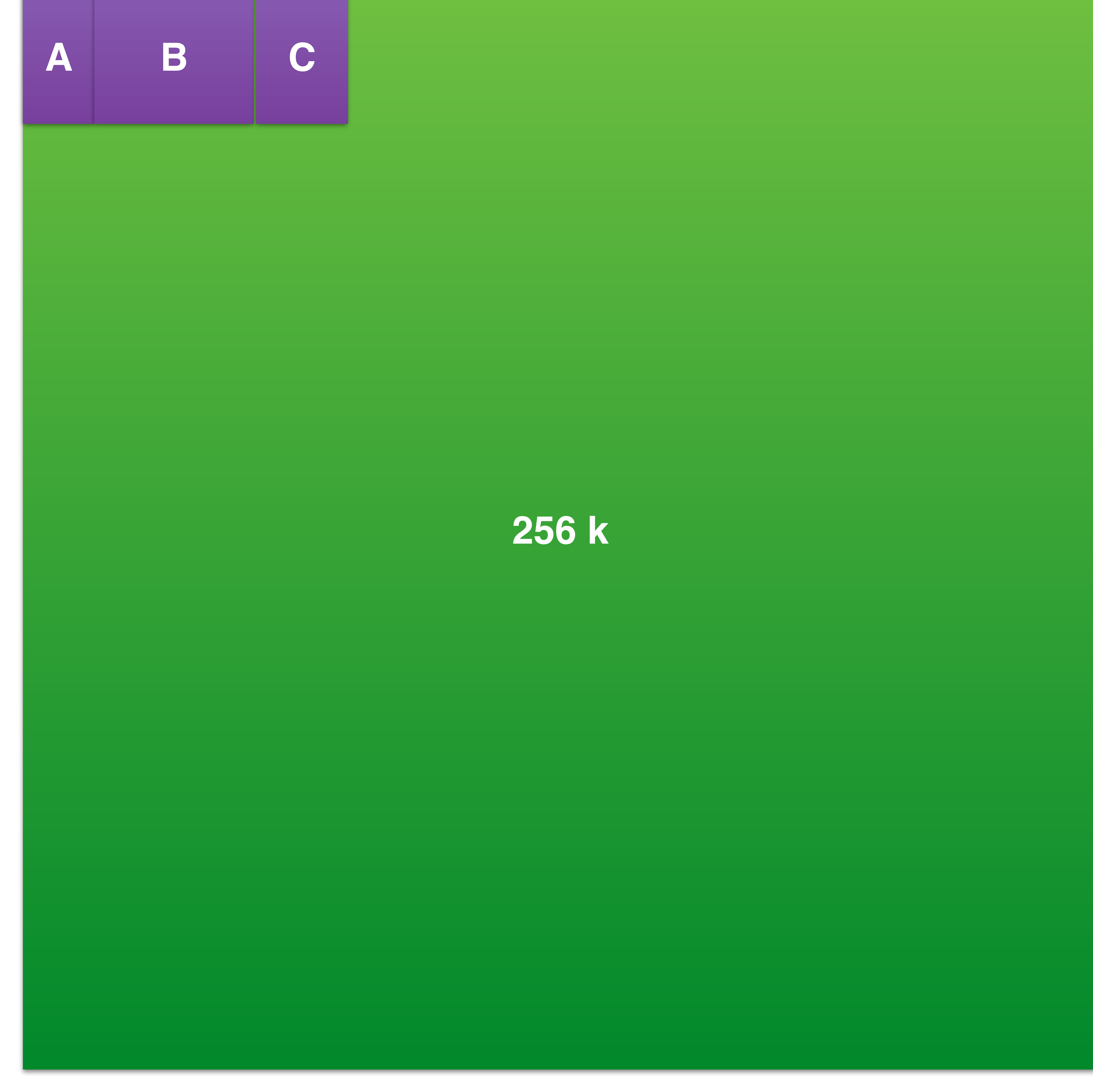
A

256 k

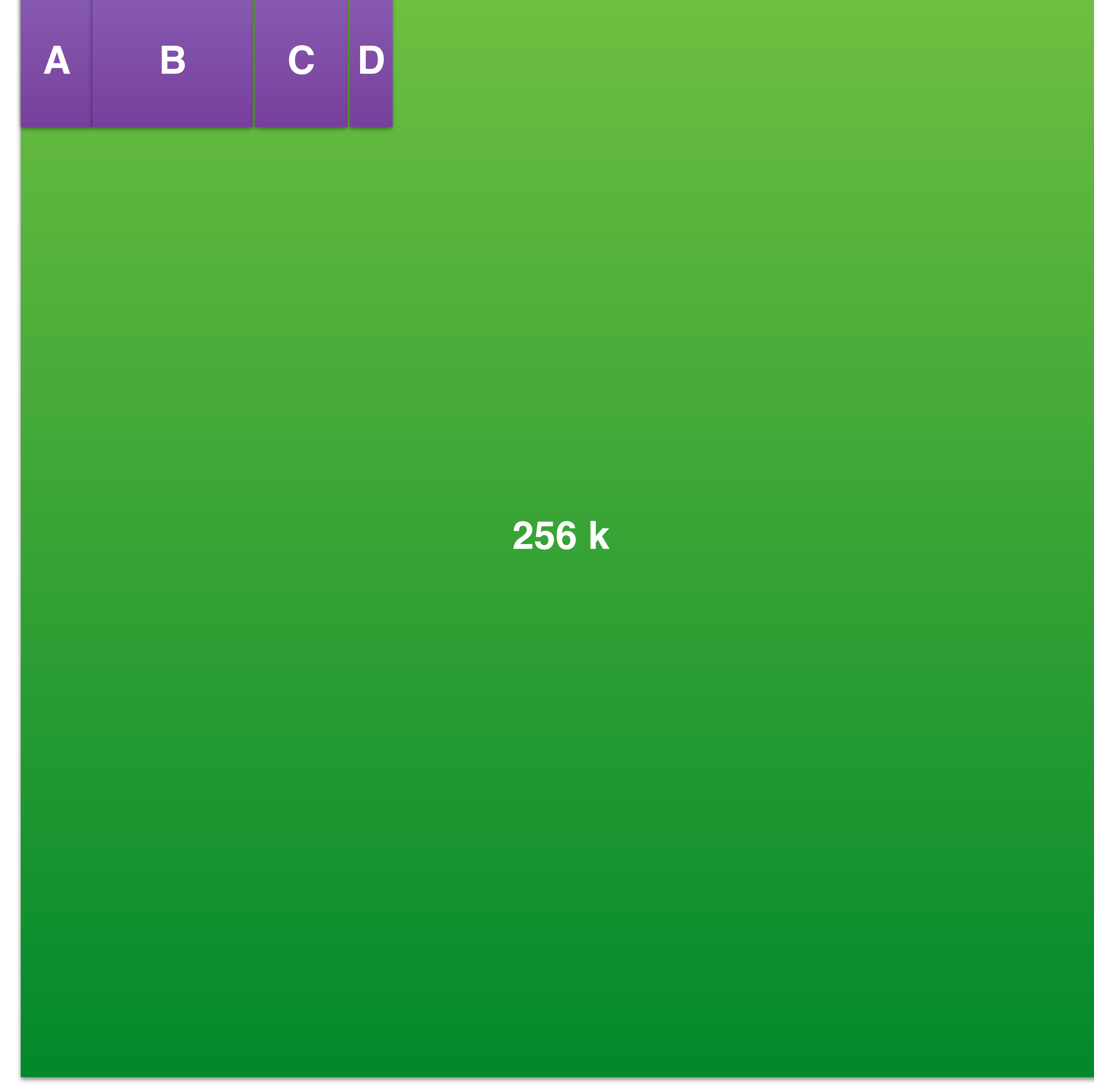
```
a = malloc(A);

b = malloc(B);
```



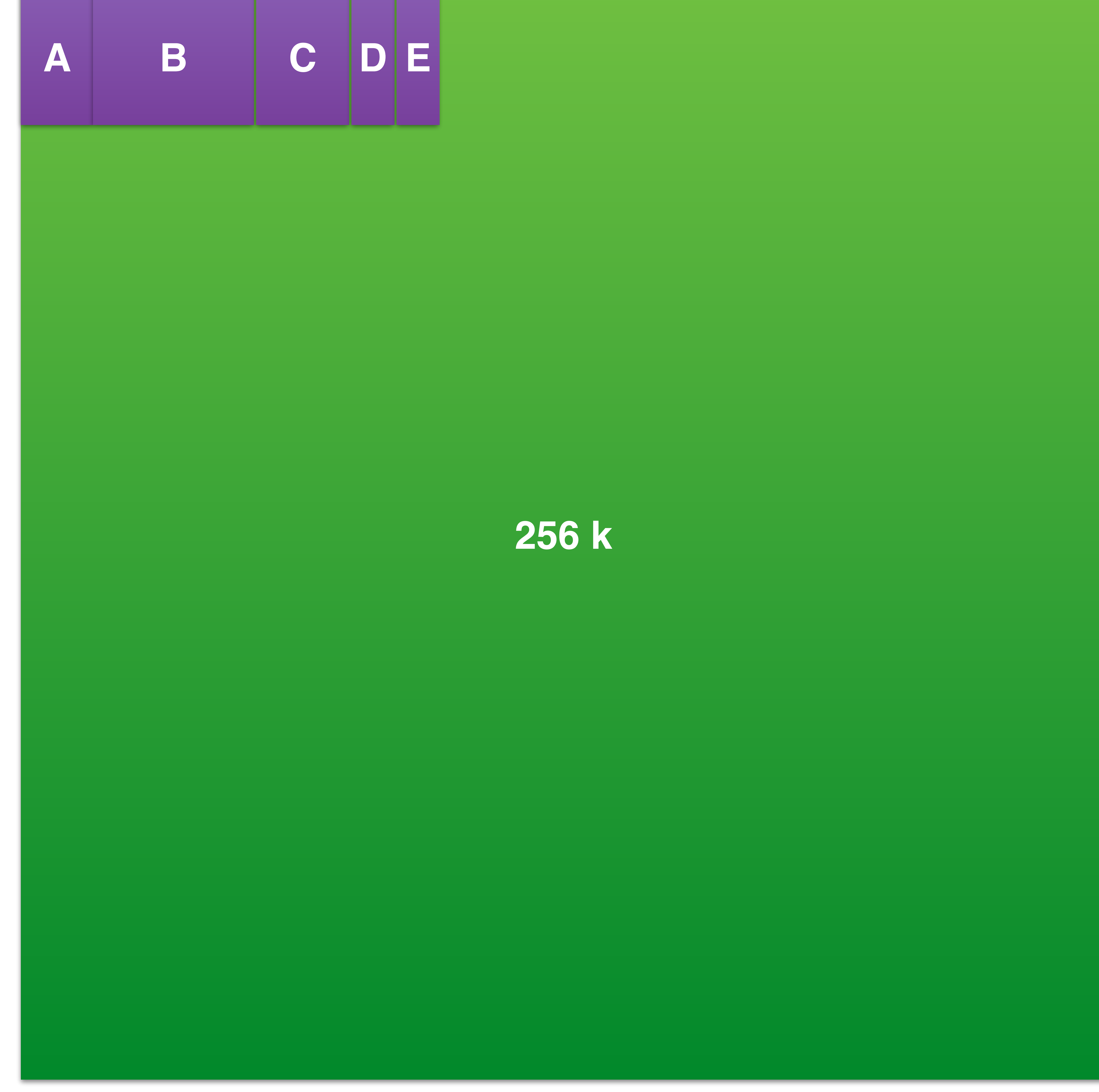
```
a = malloc(A);

b = malloc(B);

c = malloc(C);
```



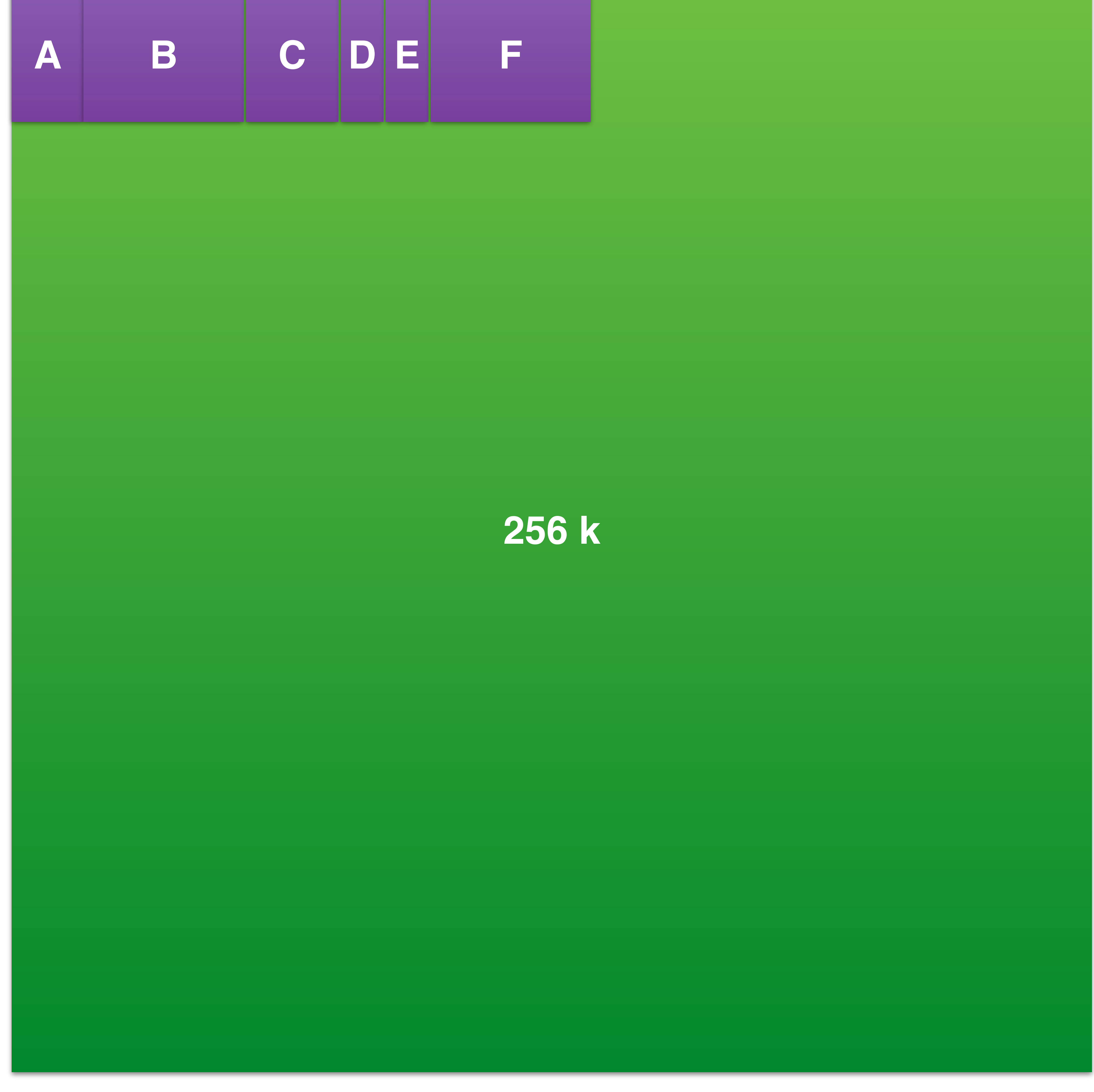
```
a = malloc(A);

b = malloc(B);

c = malloc(C);

d = malloc(D);
```



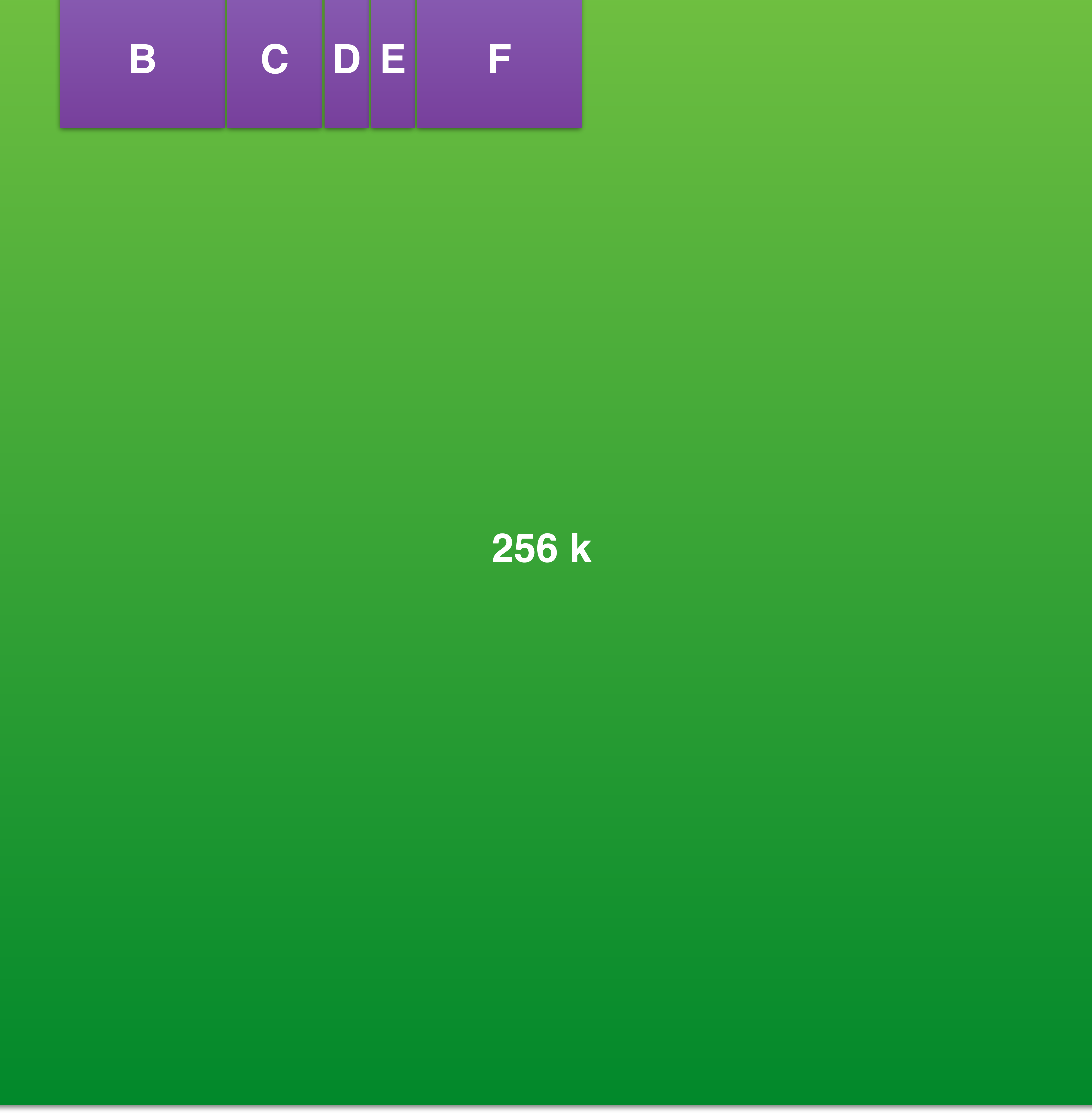
```
a = malloc(A);

b = malloc(B);

c = malloc(C);

d = malloc(D);

e = malloc(E);
```



```
a = malloc(A);
b = malloc(B);
c = malloc(C);
d = malloc(D);
e = malloc(E);
f = malloc(F);
```



```
a = malloc(A);

b = malloc(B);

c = malloc(C);

d = malloc(D);

e = malloc(E);

f = malloc(F);

free(a);
```

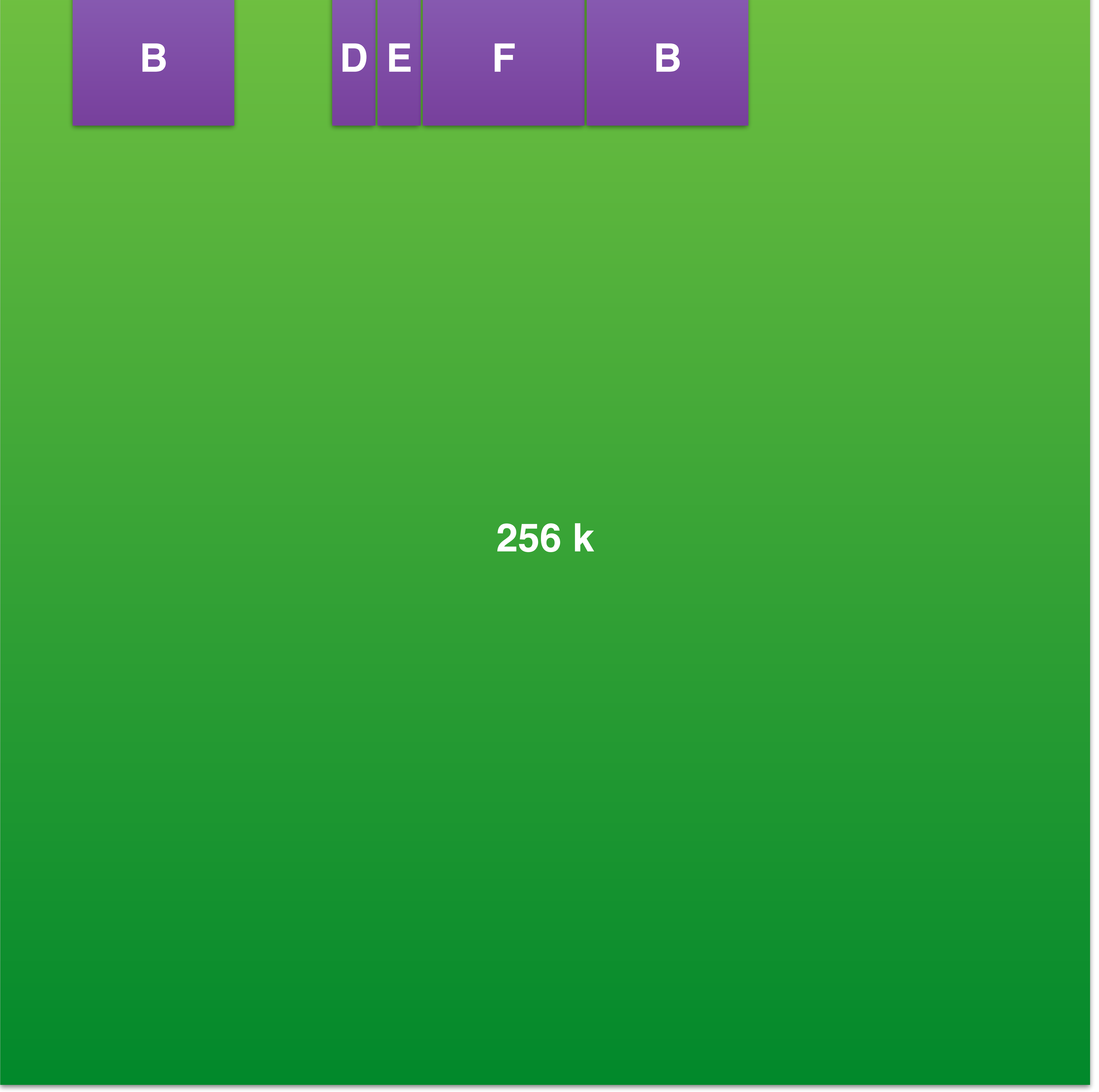


```
a = malloc(A);

b = malloc(B);

c = malloc(C);

d = malloc(D);

e = malloc(E);

f = malloc(F);

free(a);

free(c);
```

```
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);
```

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);
```



B

D

E

F

B

256 k

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

Bucket Allocation

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

Vad får detta för effekt map fragmentering?

4k	4k	4k	4k	4k	4k	4k	4k
4k	4k	4k	4k	4k	4k	4k	4k

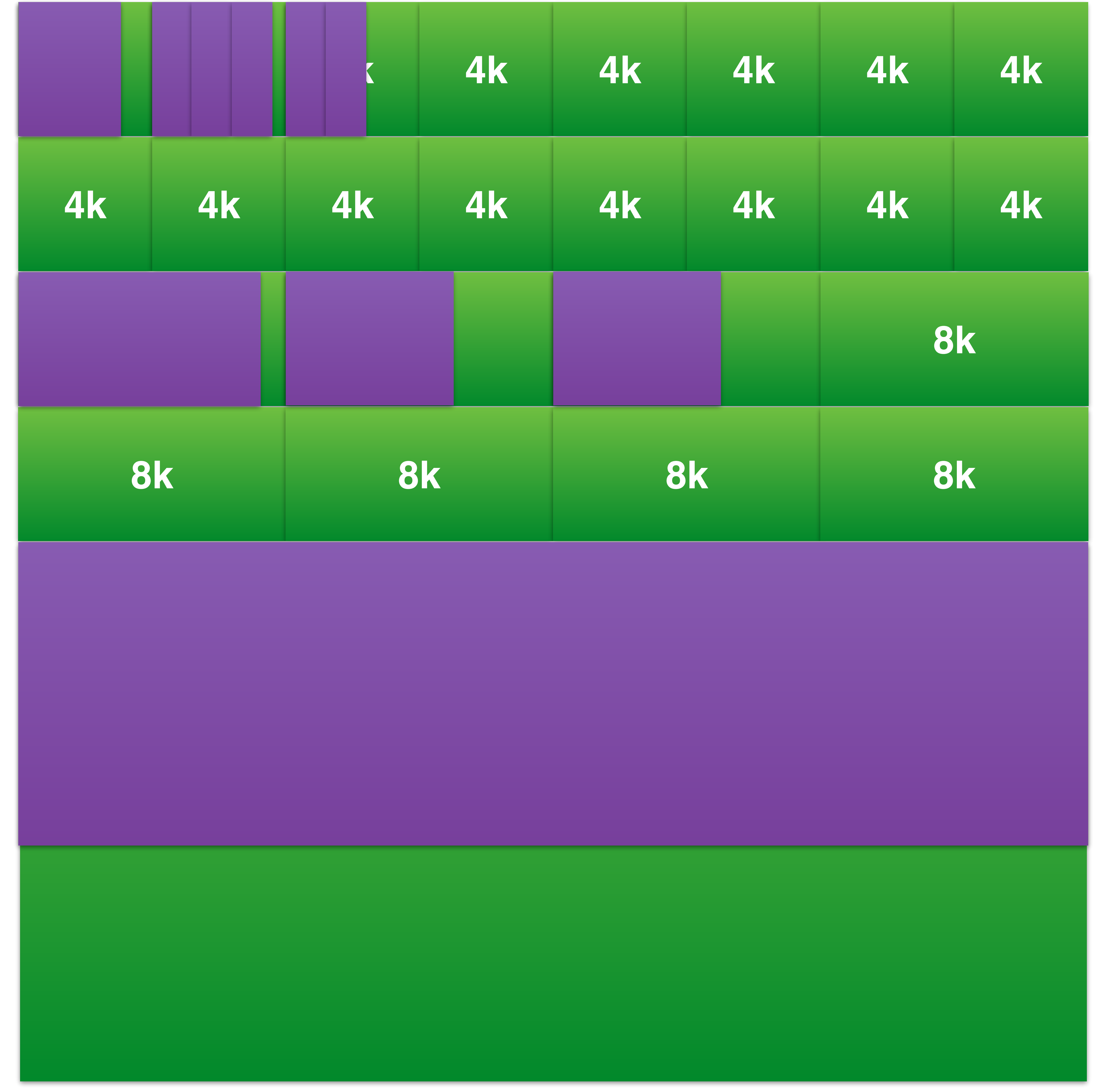
8k	8k	8k	8k
8k	8k	8k	8k

128 k

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 allokera 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

Väl mött på labben!