Föreläsning 0

Tobias Wrigstad
Kursansvarig

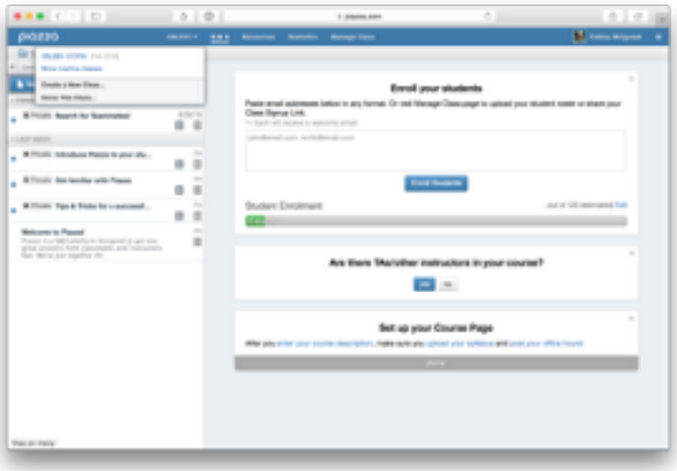
Välkommen, nu kör vi!

All information, kursmaterial

wrigstad.com/ioopm18

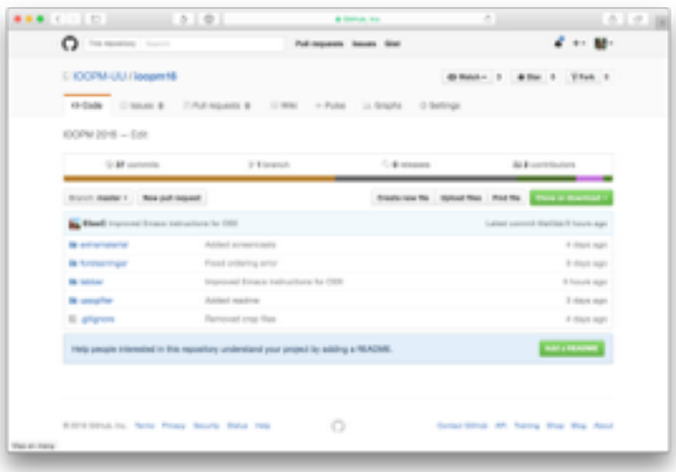
Diskussionsforum, handledning, enkäter, etc.

piazza.com/class/jkd32onglxi2vb



Utdelad kod, andra resurser

github.com/IOOPM-UU/ioopm18



Imperativ- och objektorienterad programmeringsmeto

- Du kan ”tänka programmering” — efter PKD

  Funktionell programmering

  Algoritmer, datastrukturer

- Denna fortsättningskurs fokuserar på de två vanligaste paradigmen — imperativ- & OOP

  Vi skall lära oss delar av C och Java, skriva ett par 1000 rader kod (förhoppningsvis)

  Denna vecka och nästa har vi 6 labbar i C i syfte att du skall komma igång

  Introduktion till kursen kommer föreläsning 4

todik

- IOOPM i siffror

  ~30 föreläsningar

  ~30 labbar

  4 inlämningsuppgifter

  1 projektuppgift

  2 kodprov

  30–70 olika delmål som skall redovisas var för sig, ca 2–3 i veckan

  ~12 assistenter

  ~12 timmar schemalagd undervisning varje vecka — **minst** lika mycket till krävs

  …

20 HP = 533 arbetstimmar

Kursens första 2 veckor

Haskell, fast C
(Rekursion, Generalisering, Funktionspekare)

Kom igång med C
(loopar, strängar, tal, arrayer)

Äntligen händer det något!
(I/O och funktioner)

	Mån	Tis	Ons	Tors	Fre
V35	Föreläsning Labb		Labb	Föreläsning Labb	
V36	Föreläsning Labb		Labb	Föreläsning Labb	

Intro till kursen!

Data och strukturer
(pekare och struktar)

Miniprojekt
(Använd allt tillsammans)

Buffert
(Svårt att planera utan en!)

Föreläsning 1–3

Tobias Wrigstad
Kursansvarig

Grundläggande datatyper, deklaration, uttryck och satser

Vad är imperativ programmering?

Det imperativa programspråket C

- Maskinnära språk (för någon maskin iaf)

  Resurskritiska applikationer, hårdvarunära programmering, effektivitet

- Skapades ca 1969, användes för att implementera UNIX

- Språk som kan ersätta C: C++, D, Go, Java, Rust

  På denna kurs använder vi C för att det inte gömmer komplexitet

  (Och för att det är underbart och fantastiskt!)

Några skillnader mellan C och Haskell

- C är **imperativt** och **eager** (”ivrigt”), Haskell är funktionellt och lazy
- Språken tillhör olika **syntax**familjer
- C är **manifest typat**: alla variabler måste ges en explicit typ av programmeraren
- C är **svagt typat**: vissa typomvandlingar görs automatiskt och okontrollerade brutala typomvandlingar tillåts
- C är betydligt mer **låg-nivå**:

  t.ex. C har ingen list-typ

  du kan arbeta direkt med minnesadresser (pekare)

  Minneshanteringen i C måste ofta göras explicit

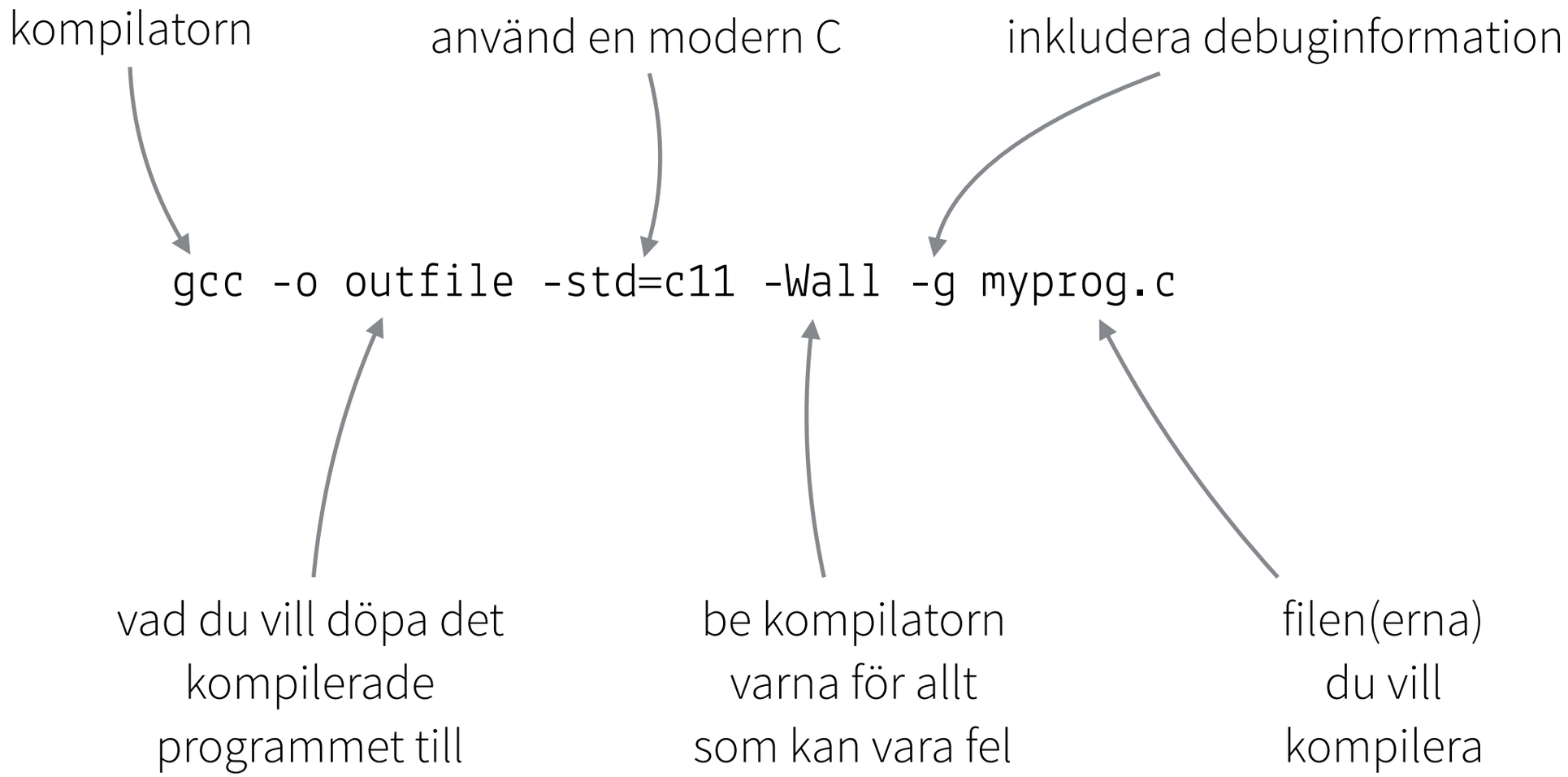
  Det görs vanligen **ingen** runtime-**kontroll** när C-program exekverar (vild adressering, arraygränser, odefinierade variabelvärden . . .)

```c
#include<stdio.h>

int main(int argc, char *argv[])
{
  puts("Hello, world!");
  return 0;
}
```

hello.c

```
$ gcc hello.c
$ ./a.out
Hello, world!
$ _
```

Kompilera ditt program



Kör ditt program: `./outfile`

Variabeldeklaration

Syntax:

```
typ variabelnamn;
typ variabelnamn = expr;
```

Exempel:

```
int age;
int age = 42;
```

- Variabler är symboliska namn för värden

  Namnet är extremt viktigt för det ger mening för programmeraren

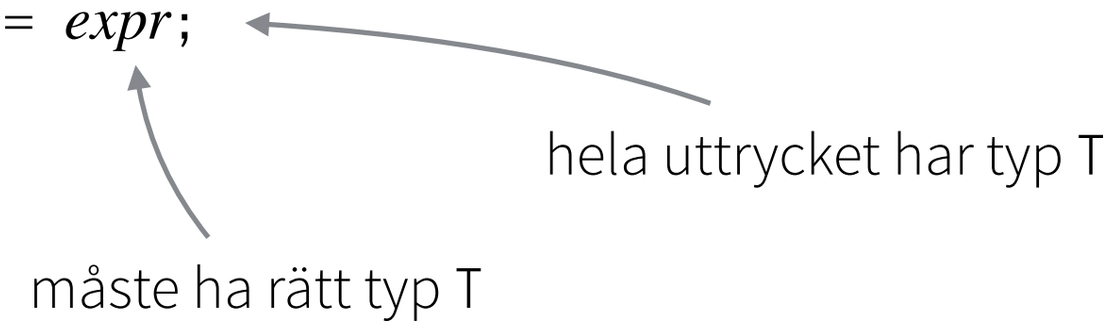
  Variabers värde kan **förändras**

  Oinitierade variablers värden är **odefinierade**

Tilldelning till variabler

Syntax:

```
variabelnamn = expr;
```



Exempel:

```
age = 100;                  // Tilldela 100 till variabeln age

age = age + 1;              // Öka variabelns värde med 1

total = age = age + 1; // OK, men vansinne
```

Datatyper [de vanligaste för nu]

Vanliga datatyper		
	Beskrivning	**Storlek**
`char`	Ett tecken	Minst 8 bitar
`short`	Litet heltal	Minst 16 bitar
`int`	Heltal	Minst 16 bitar
`long`	Stort heltal	Minst 32 bitar
`float`	Litet flyttal	Ospecificerat
`double`	Stort flyttal	Minst som `float`
`void`	Ingenting!	*n/a*
`bool`	Sedan C99	[true, false]

Kräver biblioteket `stdbool.h`

Storlekarna är beroende av vilken hårdvara programmet är kompilerat på/för.

https://en.wikipedia.org/wiki/C_data_types

```c
#include <stdbool.h>
#include <stdio.h>

int main(void)
{
  printf("bool          %zd\n", sizeof(bool));
  printf("char          %zd\n", sizeof(char));
  printf("short         %zd\n", sizeof(short));
  printf("int           %zd\n", sizeof(int));
  printf("long          %zd\n", sizeof(long));
  printf("long long     %zd\n", sizeof(long long));
  printf("float         %zd\n", sizeof(float));
  printf("double        %zd\n", sizeof(double));
  printf("long double   %zd\n", sizeof(long double));
  return 0;
}
```

data-type-sizes.c

```
$ gcc data-type-sizes.c
$ ./a.out
bool          1
char          1
short         2
int           4
long          8
long long     8
float         4
double        8
long double   16
$ _
```

Operatorer

Aritmetik	
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo
++	Inkrementera
--	Dekrementera

Relationer	
==	Likhet
!=	Olikhet
<	Strikt mindre än
<=	Mindre än
>	Strikt större än
>=	Större än

Logik	
&&	Och
||	Eller
!	Negation

Plus bitoperatorer — vi återkommer till dem senare i kursen

Många olika varianter av tilldelning

Kortform	Långform	Kommentar
`age += 1`	`age = age + 1`	
`age -= 1`	`age = age - 1`	
`age++`	`tmp = age; age = age +1; tmp`	*Vanlig felkälla!*
`++age`	`age = age + 1`	
`age--`	`tmp = age; age = age -1; tmp`	*Vanlig felkälla!*
`--age`	`age = age - 1`	
`age /= 2`	`age = age / 2`	
`age *= 2`	`age = age * 2`	

Villkorssatser (conditionals)

Syntax:

if (*expr*) { *expr*; }

if (*expr*) { *expr*; } **else** { *expr*; }

expr ? *expr* : *expr*

Exempel:

```
if (age > 100) { puts("Very old"); }

if (age % 2 == 0) { puts("Even"); } else { puts("Odd"); }

a < b ? b : a;
```

- Den vanligaste formen av villkorssats returnerar inget värde
- Den något kryptiska `? :`-formen har returvärde

Läsbarhet och frihet [alla dessa är semantiskt ekvivalenta]

```
if (age > 100) { puts(”Very old”); }
```

```
if (age > 100)
  {
    puts(”Very old”);
  }
```

```
if (age > 100)  {
  puts(”Very old”);
}
```

```
if (age > 100) puts(”Very old”);
```

```
if (age > 100)
  puts(”Very old”);
```

Läsbarhet: Apples #gotofail SSL bug [1/2]

```
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```

Indenteringen ljuger!

Indenteringen lyfter fram felet!

```
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```

https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1266

Läsbarhet: Apples #gotofail SSL bug [2/2]

```
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```

Inga block — fail!

Block — inget fail!

```
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
  {
    goto fail;
  }
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
  {
    goto fail;
    goto fail;
  }
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
  {
    goto fail;
  }
```

Switchsatser

Syntax:

```
switch (expr)
  {
  case literal  body;  break;
  default:  body;
  }
```

Exempel:

```
switch (n)
  {
  case 0:  puts("n=0"); break;
  case 1:  puts("n=1"); break;
  default: puts(”n<0 or n>1”);
  }
```

- Vanlig felkälla — bevisat dålig design

Exempel på en trasig switchsats

```
switch (n)
  {
  case 0:  puts("n=0");
  case 1:  puts("n=1");
  default: puts("n<0 or n>1");
  }
```

- Vad händer om `n == 1`?

Iteration med loopar: while

Syntax:

while (*cond*) { *body* }

while (*cond*) *expr;*

Om sant, gå ett ”till varv” i loopen

”Loop-kroppen” — det som körs varje ”varv”

Exempel:

```
int n_fakultet = 1;
int n = 6;

while (n >= 1)
  {
    n_fakultet *= n;
    n = n - 1;
  }

printf(”%d! = %d\n”, n, n_fakultet);
```

Loopar är bekväma och nödvändiga

```
// n == 6
while (n)
  n_fakultet *= n--;
```

Vad du skrev

```
n_fakultet *= n--;
n_fakultet *= n--;
n_fakultet *= n--;
n_fakultet *= n--;
n_fakultet *= n--;
n_fakultet *= n--;
```

Vad kompilatorn gjorde (unrolling)
(bara möjligt om n == 6 går att avgöra)

Loopar är bekväma och nödvändiga

```
// n == 6
while (n)
  n_fakultet *= n--;
```

Vad du skrev

```
n_fakultet *= 6;
n_fakultet *= 5;
n_fakultet *= 4;
n_fakultet *= 3;
n_fakultet *= 2;
n_fakultet *= 1;
```

Vad kompilatorn gjorde (unrolling)
(bara möjligt om n == 6 går att avgöra)

Läsbarhet [identiska satser enligt kompilatorn]

```
while (n >= 1)
  {
    n_fakultet *= n;
    n = n - 1;
  }
```

```
while (n)
  {
    n_fakultet *= n--;
  }

while (n) n_fakultet *= n--;

while (n)
  n_fakultet *= n--;
```

Kort utvikning: do–while

```
do
  {
    n_fakultet *= n;
  }
while (n--);
```

Iteration med loopar: for

Syntax:

for (*init* ; *pre* ; *post*) { *body* }

for (*init* ; *pre* ; *post*) *expr* ;

Deklarera och initiera loopvariabler

Om sant, gå ett ”till varv” i loopen

Utförs alltid sist i varje varv

Exempel:

```
int n_fakultet = 1;
for (int i = 1; i <= n; i = i + 1)
  {
    n_fakultet *= i;
  }

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

Main – där alla C-program börjar

Antalet kommandorads-argument

De faktiska kommandorads-argumenten

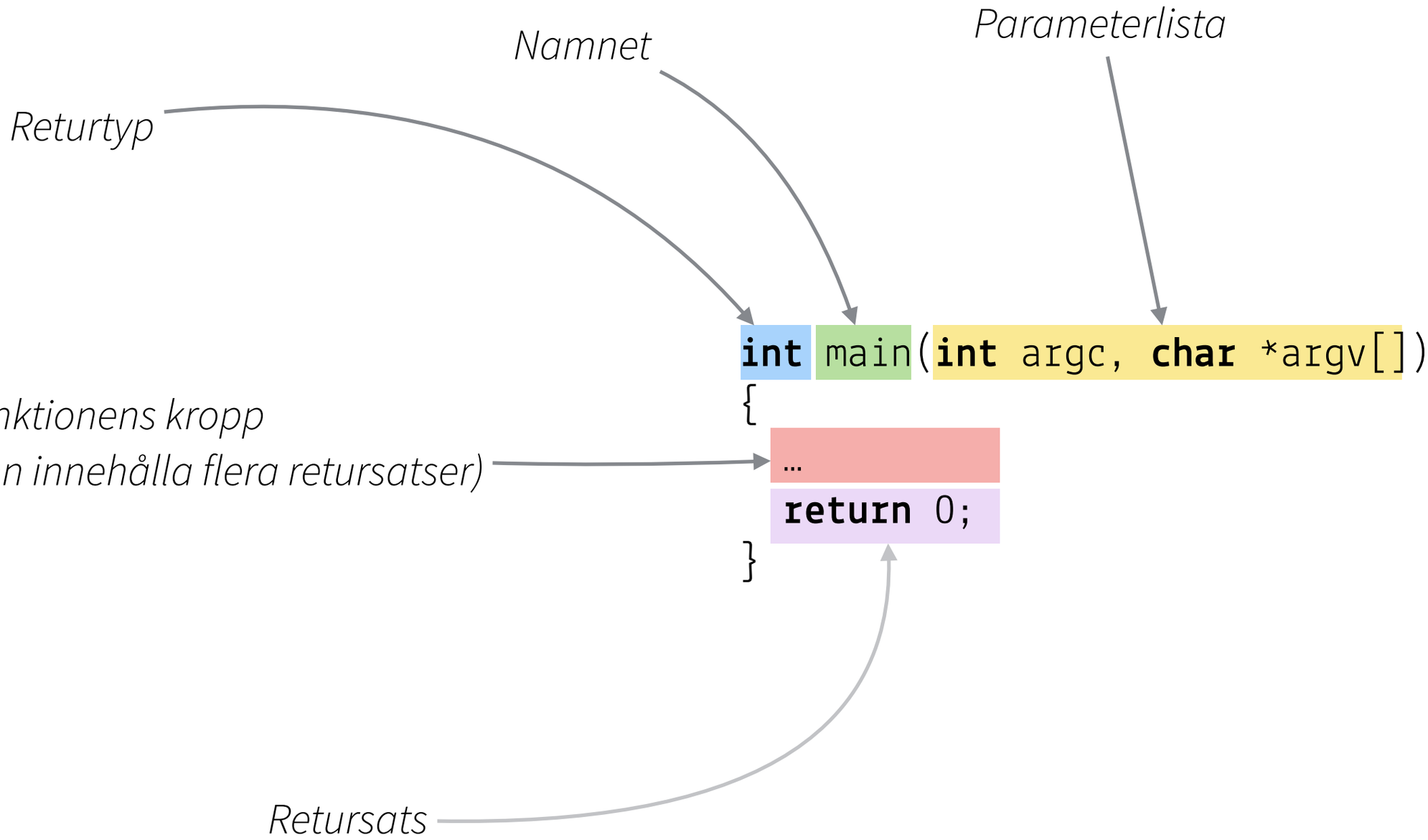
```
int main(void)
{
  …
  return 0;
}
```

OK

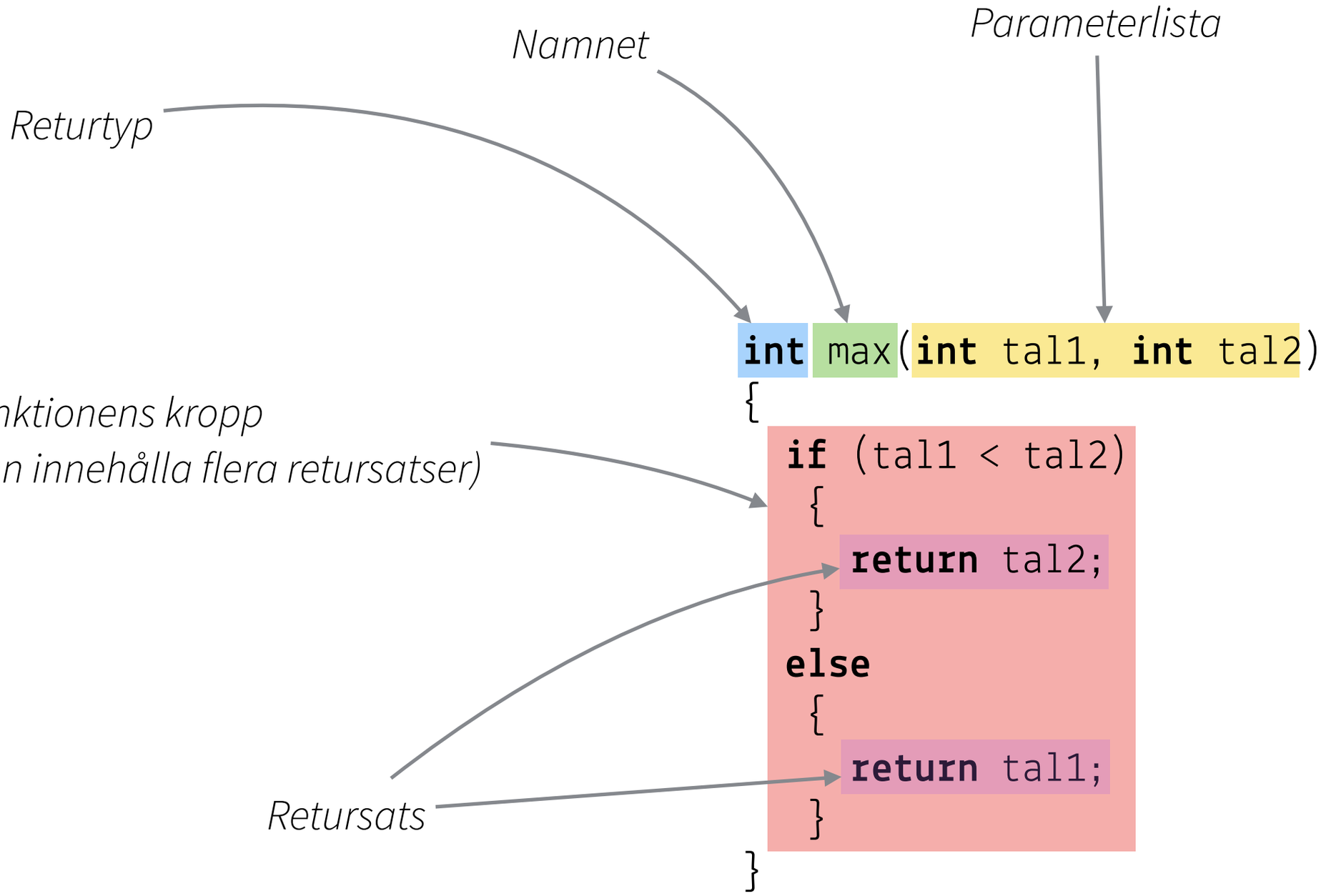
```
int main(int argc, char *argv[])
{
  …
  return 0;
}
```

Bättre

Funktionens anatomi



Deklarera egna funktioner



Deklarera egna funktioner

```
// Exempel på anrop
int a = max(512, 1024);
```

```
int max(int tal1, int tal2)
{
  if (tal1 < tal2)
   {
     return tal2;
   }
  else
   {
     return tal1;
   }
}
```

Inkludera funktioner från andra bibliotek

```
#include <filnamn.h>
```
← Inkludera från standardbibliotek

```
#include "filnamn.h"
```
← Inkludera från ditt eget program

Plus extra länkning i kompileringssteg. Vi återkommer till det senare.

Exempel på olika funktioner

Funktion	Kommentar
`void puts(char *)`	Skriv ut en sträng på skärmen
`int atoi(char *)`	Konvertera en sträng till ett heltal (int)
`long atol(char *)`	Konvertera en sträng till ett heltal (long)
`int getchar()`	Läs in ett tecken från tangentbordet
`FILE *fopen(char *, char *)`	Öppna en fil

Läs mer om funktionerna med hjälp av `man`-kommandot

Läsbarhet [identiska funktioner enligt kompilatorn]

```
int max(int tal1, int tal2)
{
  if (tal1 < tal2)
   {
     return tal2;
   }
  else
   {
     return tal1;
   }
}
```

```
int max(int tal1, int tal2)
{
  return (tal1 < tal2) ? tal2 : tal1;
}
```

```
int max(int tal1, int tal2)
{
  if (tal1 < tal2) return tal2;
  return tal1;
}
```

Returns kontrollflöde [vanlig felkälla]

```
int max(int tal1, int tal2)
{
  if (tal1 < tal2)
   {
     return tal2;
   }
  else
   {
     return tal1;
   }

  puts(”Jag skrivs aldrig ut!”);
}
```

Program som skriver ut kommandoradsargument

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

int main(int argc, char *argv[])
{
  printf("%d kommandoradsargument\n", argc);
  for (int i = 0; i < argc; ++i)
    {
      printf("Argument %d = %s\n", i, argv[i]);
    }
  return 0;
}
```

echo.c

Komplett exempel för n-fakultet

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

int main(int argc, char *argv[])
{
  int nfak = 1;
  int n = atoi(argv[1]);
  for (int i = 1; i <= n; ++i)
    {
      nfak *= i;
    }
  printf("%d! = %d\n", n, nfak);
  return 0;
}
```

n-fakultet.c

Arrayer

```
T name[size]; // deklarerar en array av size element av typen T

int salaries[500];
long sum = 0;

for (int i = 0; i < 500; ++i)
  {
    sum += salaries[i];
  }
```

läs det i:te elementet i arrayen

- Arrayer har en fix storlek – kan inte ändras
- Arrayer indexeras [0, ***size***) – första elementet har index 0, sista ***size***−1

  Skriv: `myarr[17] = 42;` Läs: `myarr[x]`

- Arrayerna har inget metadata, och C gör ingen indexkontroll

Ingen indexkontroll

```
int salaries[500];
long sum = 0;

for (int i = 0; i <= 500; ++i)
  {
    sum += salaries[i];
  }
```

Vad blir resultatet av detta program när det körs?

Ingen indexkontroll

```
int salaries[500];
long sum = 0;

for (int i = 0; i <= 500; ++i)
  {
    sum += salaries[i];
  }
```

UNDEFINED BEHAVIOUR

Vad blir resultatet av detta program när det körs?

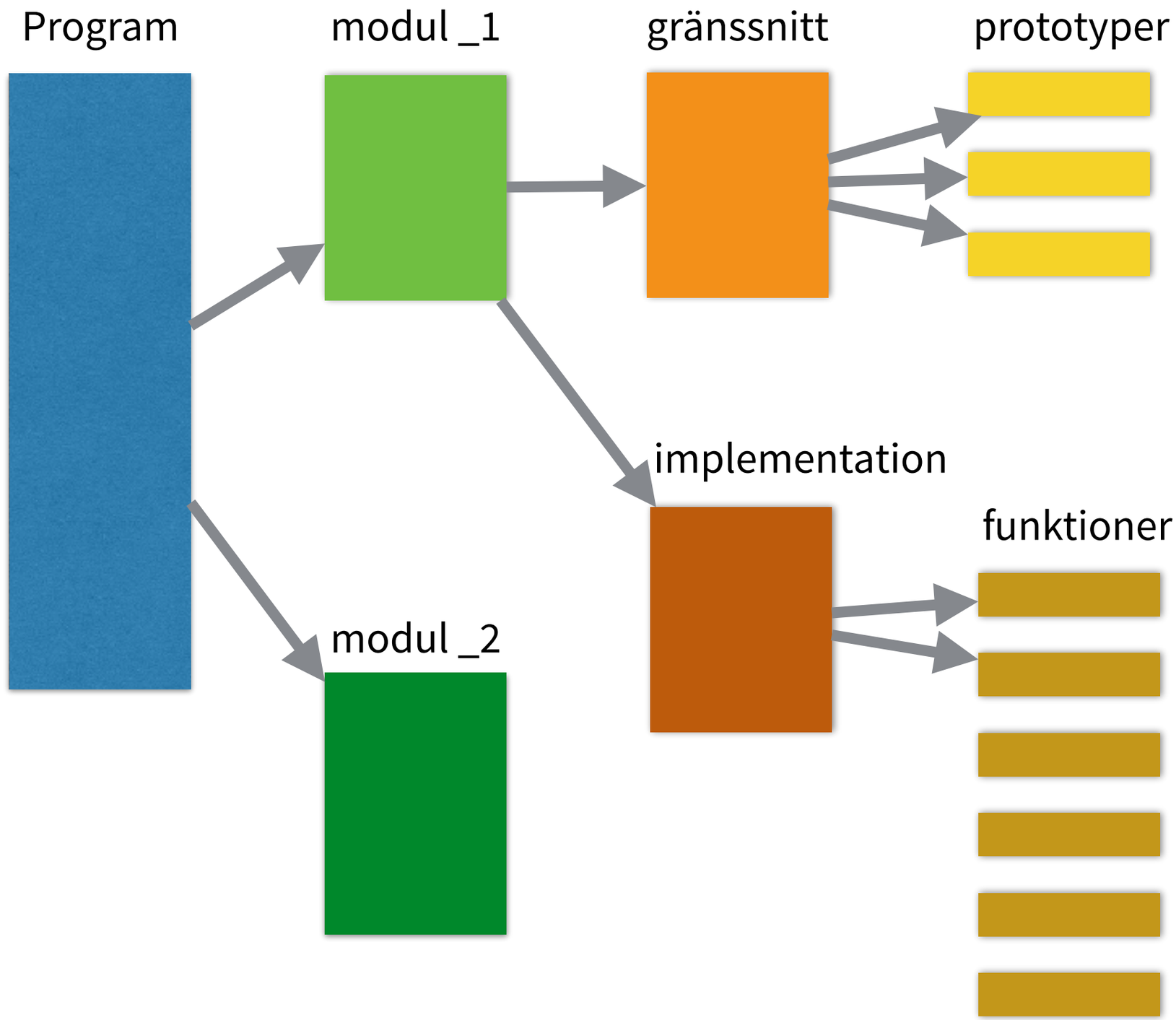
Kommentarer

```
// Startar kommentar som gäller till radens slut

/* Startar kommentarblock som gäller ända till */
```

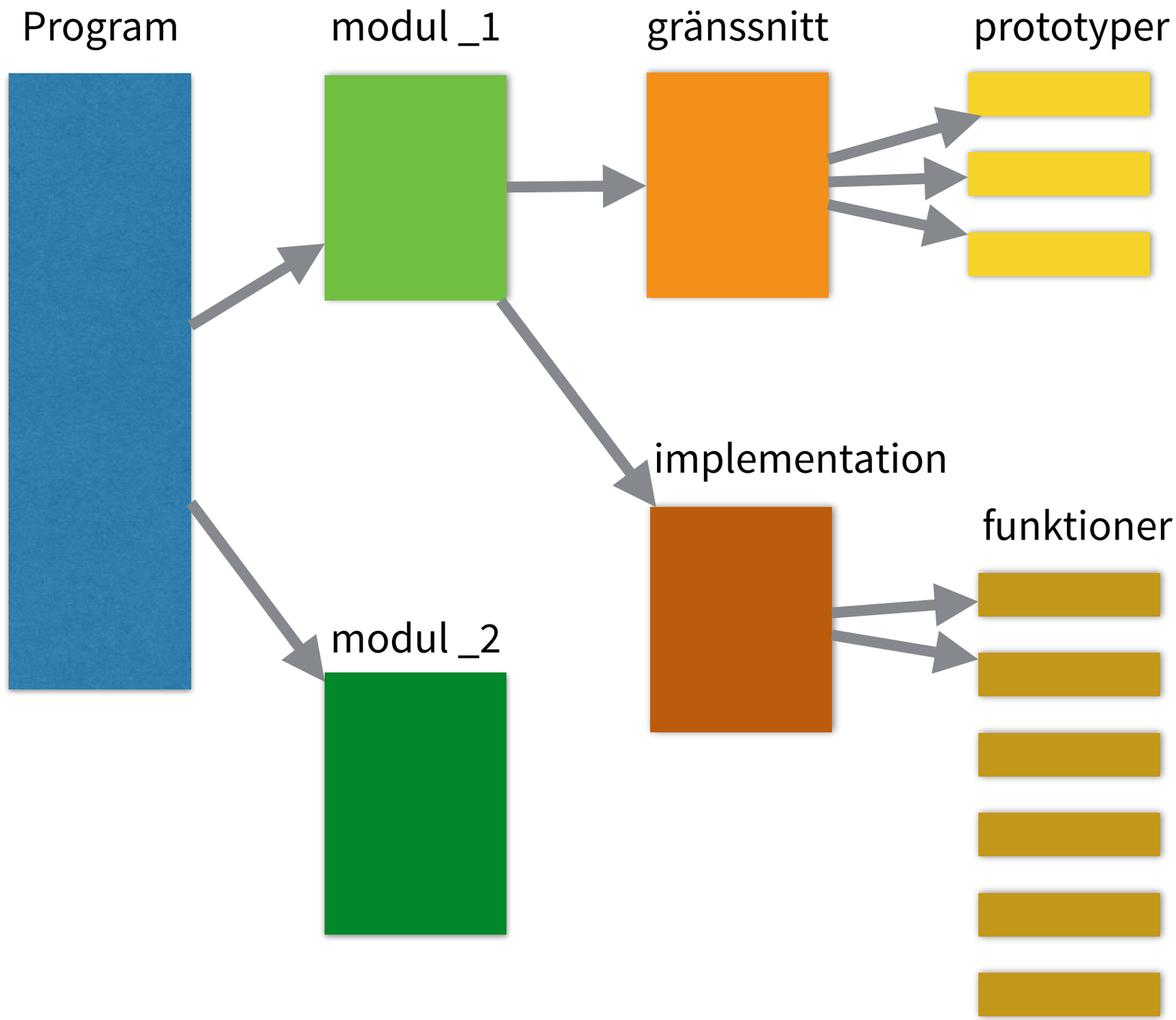
Kommentarer är mest nödvändiga för att förklara *varför.*

Om du känner att du behöver kommentera en bit kod för att den skall gå att förstå är det 99% chans att koden borde skrivas om istället för kommenteras.

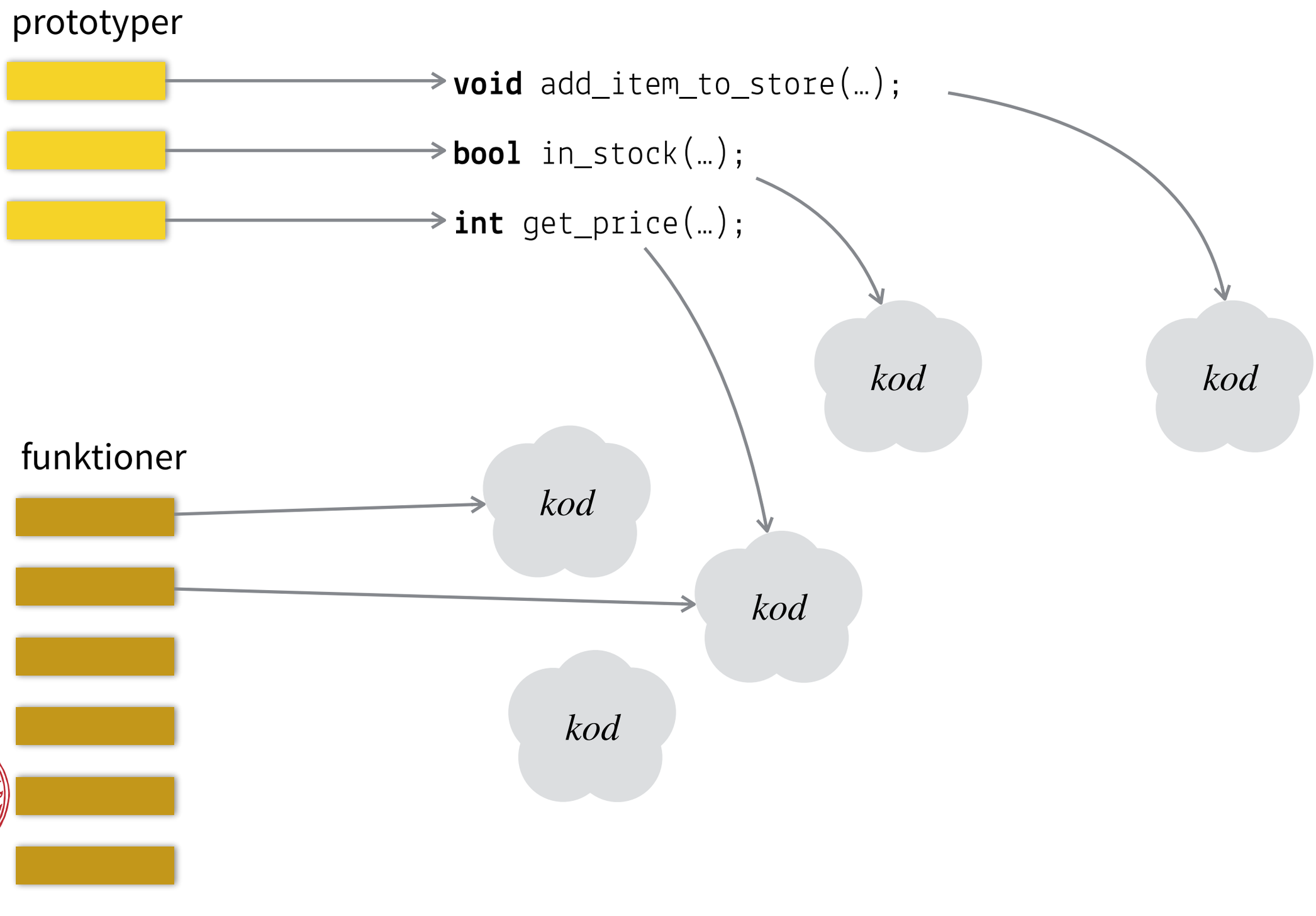
Ett programs anatomi



Ett programs anatomi



Funktionsabstraktioner



Funktionsabstraktionen

Bygga abstraktioner av abstraktioner!

```
int make_purchase(db_t db, cart_t cart)
{
  int bill = 0;

  for (int i = 0; i < cart.size; ++i)
    {
      if (!in_stock(cart.item[i])
        {
          remove_from_cart(cart, i);
        }
      else
        {
          bill += get_price(cart.item, db);
        }
    }

  return bill;
}
```

kod

kod

kod

Föreläsning 1–3

Gås endast igenom kursivt på föreläsningen och i mån av tid — dock bra att ha tillgängligt inför lab 2 & 3

Introduktion till standard-I/O och grundläggande stränghantering

Hur var det i Haskell?

- Char är en egen typ

```
'a' :: Char
```

```
'a' + 2          — kompilerar ej
```

Hur var det i Haskell?

- Char är en egen typ

```
'a' :: Char

'a' + 2          — kompilerar ej
```

- **type String** = [**Char**]

```
"Hello" == ['H', 'e', 'l', 'l', 'o']

'F' : "oo" ++ "Bar!" == "FooBar!"
```

Hur var det i Haskell?

- Char är en egen typ

```
'a' :: Char

'a' + 2                    — kompilerar ej
```

- **type String** = [**Char**]

```
"Hello" == ['H', 'e', 'l', 'l', 'o']

'F' : "oo" ++ "Bar!" == "FooBar!"
```

- Kan du listor kan du strängar! (mer eller mindre)

Tecken i C

- **`char`** — ett heltal (på *normalt* en byte)
- Ett tecken är en **`char`** med motsvarande ASCII-värde

```
char c1 = 'a';
char c2 = 97;          == c1
char c3 = 'a' + 2;     == 'c'
char c4 = '2' + '2'; == 50 + 50
```

Tecken i C

- **`char`** — ett heltal (på *normalt* en byte)
- Ett tecken är en **`char`** med motsvarande ASCII-värde

```
char c1 = 'a';
char c2 = 97;          == c1
char c3 = 'a' + 2;     == 'c'
char c4 = '2' + '2'; == 100
```

Tecken i C

- **`char`** — ett heltal (på *normalt* en byte)
- Ett tecken är en **`char`** med motsvarande ASCII-värde

```
char c1 = 'a';
char c2 = 97;          == c1
char c3 = 'a' + 2;     == 'c'
char c4 = '2' + '2'; == 'd'
```

Tecken i C

- **`char`** — ett heltal (på *normalt* en byte)
- Ett tecken är en **`char`** med motsvarande ASCII-värde

```
char c1 = 'a';
char c2 = 97;          == c1
char c3 = 'a' + 2;     == 'c'
char c4 = '2' + '2'; == 'd'
```

- Obs! `'a'` och 97 är alltså ekvivalenta

  `'a'` är *syntaktiskt socker* för 97

Tecken i C

```c
#include <stdio.h>

int main(void)
{
  for (int c = 0; c < 128; c++)
    {
      printf("Nr: %d, Char: %c\n", c, c);
    }

  return 0;
}
```

Är c en int (%d) eller ett tecken (%c)?

ascii.c

Strängar i C

- C har ingen strängtyp!

  `char*` — en pekare till en array av chars

  `char[]` — en array av chars

  I stora stycken är typerna ovan ekvivalenta

Strängar i C

- C har ingen strängtyp!

  `char*` — en pekare till en array av chars

  `char[]` — en array av chars

  I stora stycken är typerna ovan ekvivalenta

- Kan du listor kan du strängar! (mer eller mindre)

  `char s[] = "Hello"` är **precis** samma sak som

  `char s[] = {'H', 'e', 'l', 'l', 'o', '\0'}` vilket är **precis** samma sak som

  `char s[] = {72, 101, 108, 108, 111, 0}`

Strängar i C

- C har ingen strängtyp!

  `char*` — en pekare till en array av chars

  `char[]` — en array av chars

  I stora stycken är typerna ovan ekvivalenta

- Kan du listor kan du strängar! (mer eller mindre)

  `char s[] = "Hello"` är **precis** samma sak som

  `char s[] = {'H', 'e', 'l', 'l', 'o', '\0'}` vilket är **precis** samma sak som

  `char s[] = {72, 101, 108, 108, 111, 0}`

  `s =`

'H'	'e'	'l'	'l'	'o'	'\0'

Strängar i C

- C har ingen strängtyp!

  `char*` — en pekare till en array av chars

  `char[]` — en array av chars

  I stora stycken är typerna ovan ekvivalenta

- Kan du listor kan du strängar! (mer eller mindre)

  `char s[] = "Hello"` är **precis** samma sak som

  `char s[] = {'H', 'e', 'l', 'l', 'o', '\0'}` vilket är **precis** samma sak som

  `char s[] = {72, 101, 108, 108, 111, 0}`

  `s =` | `'H'` | `'e'` | `'l'` | `'l'` | `'o'` | `'\0'` |

  *Hello är en array av **sex** chars!*

Vanliga misstag 1: Strängjämförelse

- Strängjämförelse med == avser *identitet*, inte *ekvivalens*

```c
#include <stdio.h>

int main (void)
{
  char password[] = "abc123";
  char entered[128];

  puts("Please enter the secret code:");
  scanf(”%s”, entered); // read input

  if (entered == password)
    {
      puts("You are logged in!");
    }
  else
    {
      puts("Incorrect password!");
    }

  return 0;
}
```

password.c

```c
#include <stdio.h>

int main (void)
{
  char password[] = "abc123";
  char *entered;        // kommer att ”peka ut” inläst data
  size_t entered_size; // längden på entered

  puts("Please enter the secret code:");
  getline(&entered, &entered_size, stdin); // read input

  if (entered == password)
    {
      puts("You are logged in!");
    }
  else
    {
      puts("Incorrect password!");
    }

  return 0;
}
```

password-bad.c

```c
#include <stdio.h>

int main (void)
{
  char password[] = "abc123";
  char *entered;        // kommer att ”peka ut” inläst data
  size_t entered_size; // längden på entered

  puts("Please enter the secret code:");
  getline(&entered, &entered_size, stdin); // read input

  if (strncmp(entered, password, entered_size) == 0)
    {
      puts("You are logged in!");
    }
  else
    {
      puts("Incorrect password!");
    }

  return 0;
}
```

password-good.c

Vanliga misstag 2

- Aliasering!

Inläsning med `getline()` undviker detta!

```c
#include <stdio.h>

int main (void)
{
  char buffer [128];
  char *first;
  char *last;

  puts("What is your first name?");
  scanf("%s", buffer);
  first = buffer;

  puts("What is your last name?");
  scanf("%s", buffer);
  last = buffer;

  printf("Hello %s %s!\n", first, last);
  return 0;
}
```

Vanliga misstag 2: Aliasering

```c
#include<stdio.h>

int main()
{
  char buffer[128];
  char first[128];
  char last[128];

  puts("What is your first name?");
  scanf("%s", buffer);
  first = buffer; // Fel! first och buffer är nu **samma* sträng

  puts("What is your last name?");
  scanf("%s", buffer); // När vi ändrar buffer här ändrar vi samtidigt first!
  last = buffer; // Fel! first, last och buffer är nu **samma* sträng

  printf("Hello %s %s!\n", first, last);
  return 0;
}
```

greeting.c

Vanliga misstag 3

- Strängar är arrayer som slutar med '\0'

```c
#include <stdio.h>

void copy(char to[], char from[], int len)
{
  while(len--)
    to[len] = from[len];
}

int main (void)
{
  char *s = "Hello";
  char t[5];

  copy(t, s, 5);

  puts(t);

  return 0;
}
```

Vanliga misstag 3: …som slutar med '\0'!

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

int main(int argc, char *argv[])
{
  char *s = "Hello";
  char t[5]; // Fel! För att få plats med "Hello" krävs 6 tecken
             // 5 för strängen **plus 1 för '\0'**
  strcpy(t, s);
  puts(t);

  return 0;
}
```

length.c

Standardbiblioteket `string.h`

- Inkluderas med `#include <string.h>`

Funktion	Beskrivning
`strlen(s)`	längden av s (utan `'\0'`-tecknet!)
`strncpy(s, t, n)`	kopiera t till s (upp till n tecken)
`strncmp(s, t, n)`	jämför s och t (upp till n tecken)
`== 0`	s och t är samma sträng
`> 0`	s kommer efter t i bokstavsordning
`< 0`	s kommer före t i bokstavsordning
`strncat(s, t, n)`	Lägg t följt av s i t

Gör `man string`

Funktioner i `stdlib.h` (togs även upp tidigare)

- Inkluderas med `#include <stdlib.h>`

  `atoi(s)` — konvertera strängrepresentation av tal till motsvarande heltal

  `atol(s)` — med `long` istf `int` som returtyp

  Exempel:

  `atoi("123abc") == 123` (som int)

OBS!

- Alla funktioner är optimistiska

  Förutsätter från att det finns tillräckligt minne i datat som används

  Förutsätter att strängar är `'\0'`-terminerade korrekt

- Använd alltid `str`**`n`**`cpy` (`str`**`n`**`cmp`, …) över `strcpy` (`strcmp`, …) etc.

OBS!

- Alla funktioner är optimistiska

  Förutsätter från att det finns tillräckligt minne i datat som används

  Förutsätter att strängar är `'\0'`-terminerade korrekt

- Använd alltid `str`**`n`**`cpy` (`str`**`n`**`cmp`, …) över `strcpy` (`strcmp`, …) etc.
- Använd alltid funktioner från standardbiblioteken (som string.h) över funktioner du skriver själv

OBS!

- Alla funktioner är optimistiska

  Förutsätter från att det finns tillräckligt minne i datat som används

  Förutsätter att strängar är `'\0'`-terminerade korrekt

- Använd alltid `strncpy` (`strncmp`, …) över `strcpy` (`strcmp`, …) etc.
- Använd alltid funktioner från standardbiblioteken (som string.h) över funktioner du skriver själv

  Även om det är en bra övning att ha implementerat motsvarande funktioner själv någon gång!

Standardbiblioteket `ctype.h`

- Inkluderas med `#include <ctype.h>`

Funktion	Beskrivning
`isalpha(c)`	Är c en bokstav?
`isdigit(c)`	Är c en siffra?
`islower(c)`	Är c en gemen? (liten bokstav)
`digittoint(c)`	Konvertera från ’2’ till 2 (som int)
`toupper(c)`	Från gemen till versal (’a’ till ’A’)

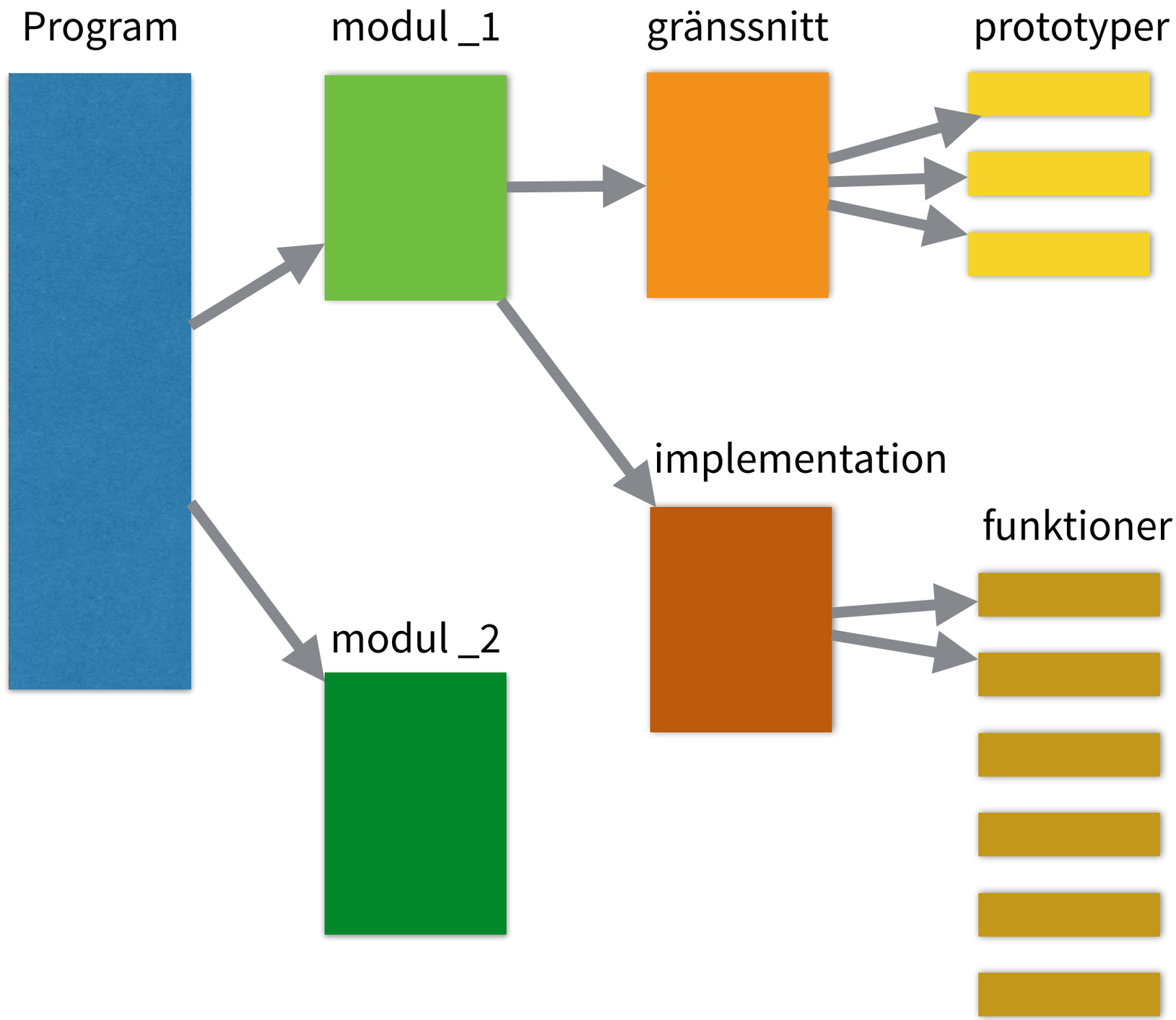
Gör `man ctype`

Föreläsning 1–3

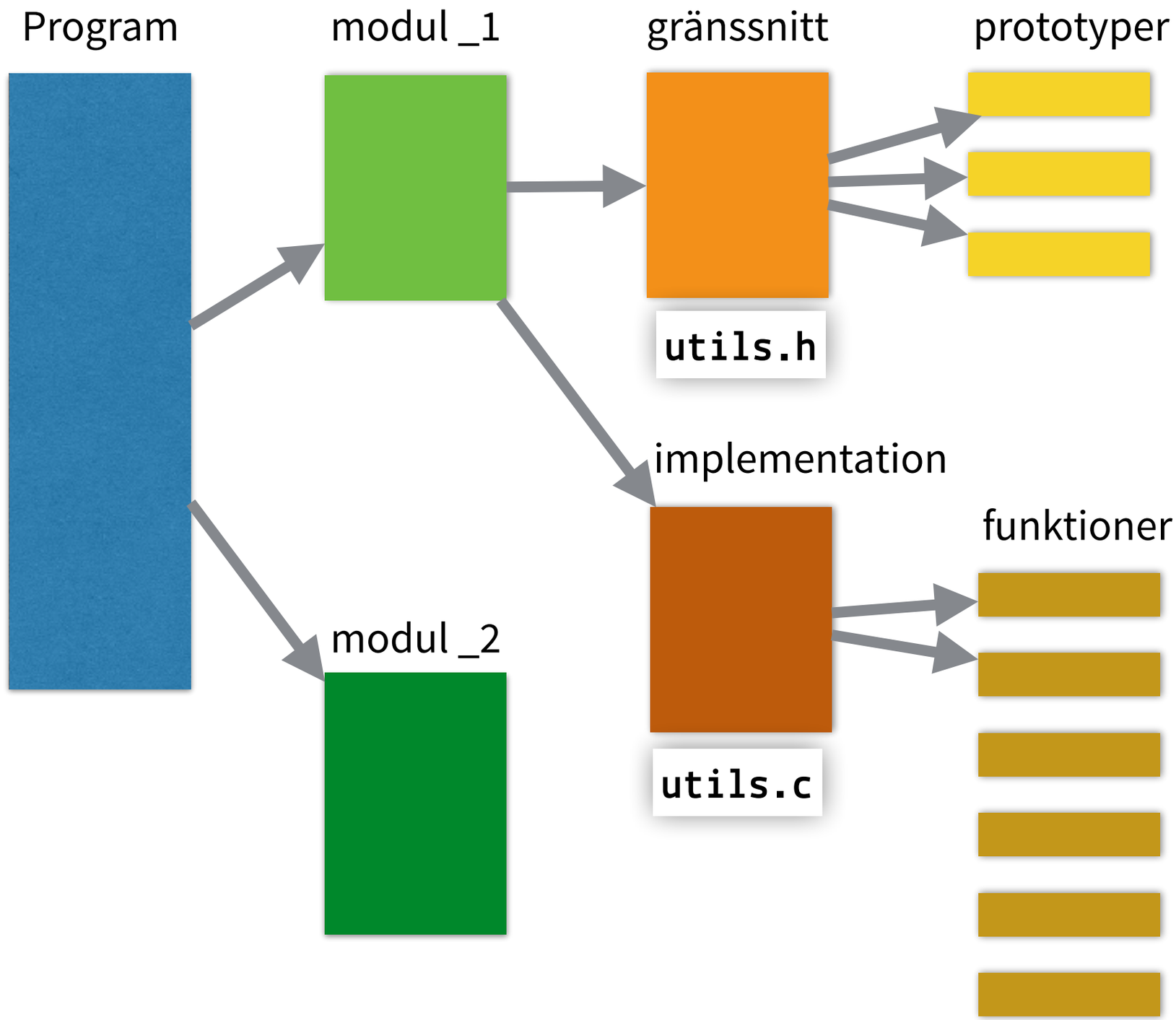
Tobias Wrigstad

Poster, unioner och typedef

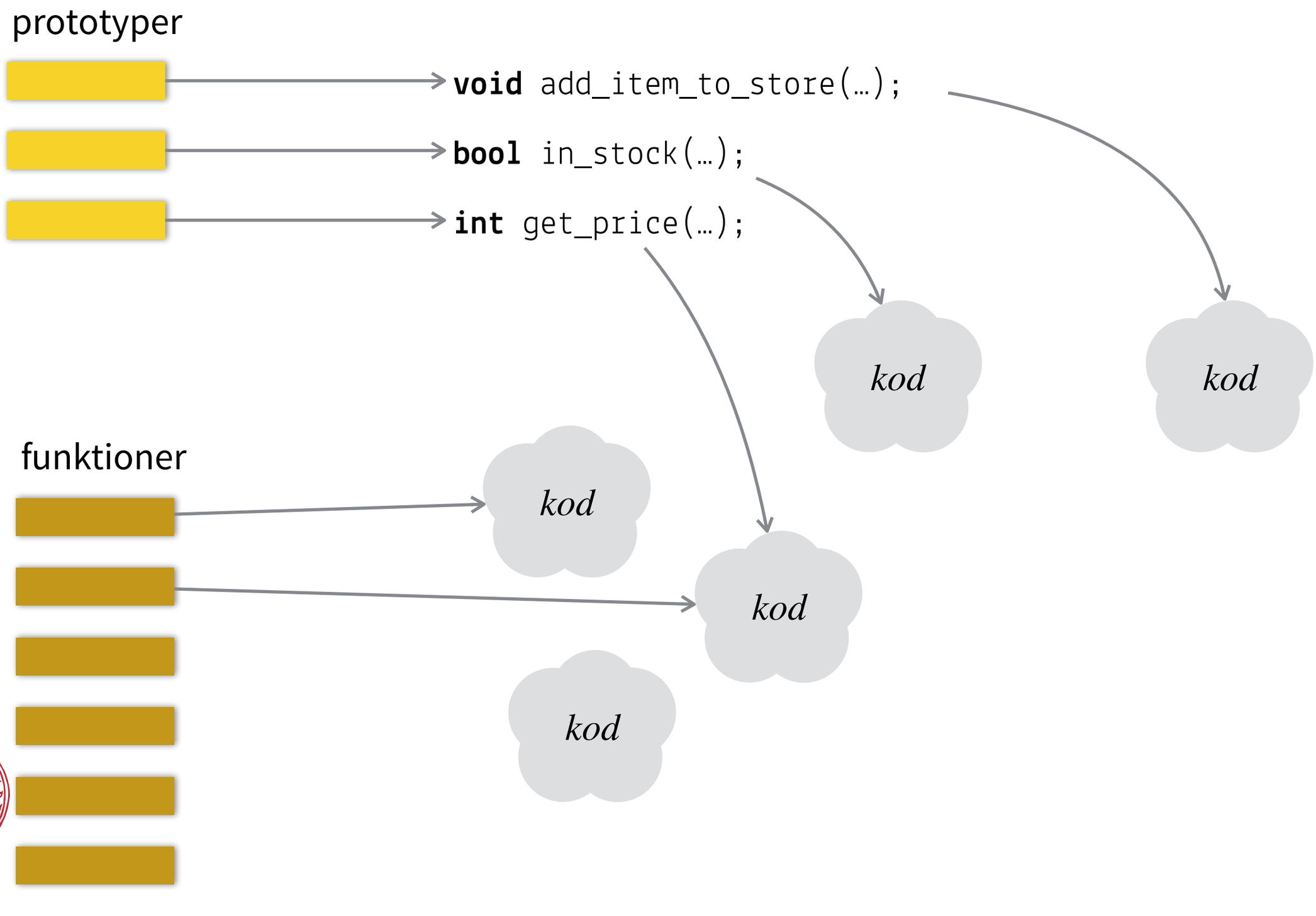
Ett programs anatomi



Ett programs anatomi



Funktionsabstraktioner



Funktionsabstraktionen

Bygga abstraktioner av abstraktioner!

```
int make_purchase(db_t db, cart_t cart)
{
  int bill = 0;

  for (int i = 0; i < cart.size; ++i)
    {
      if (!in_stock(cart.item[i])
        {
          remove_from_cart(cart, i);
        }
      else
        {
          bill += get_price(cart.item, db);
        }
    }

  return bill;
}
```

kod

kod

kod

Sammansatta datatyper

- Minns Haskell…

  Kraftfullt stöd för matchning, konstruktion och dekonstruktion av värden

  Algebraiska datatyper

```
                    -- name, year, month, day
data Anniversary = Birthday String Int Int Int
                    -- spouse name 1, spouse name 2, year, month, day
                 | Wedding String String Int Int Int

smithWedding = Wedding "John Smith" "Jane Smith" 1987 3 4
```

Sammansatta datatyper i C

värdesemantik!

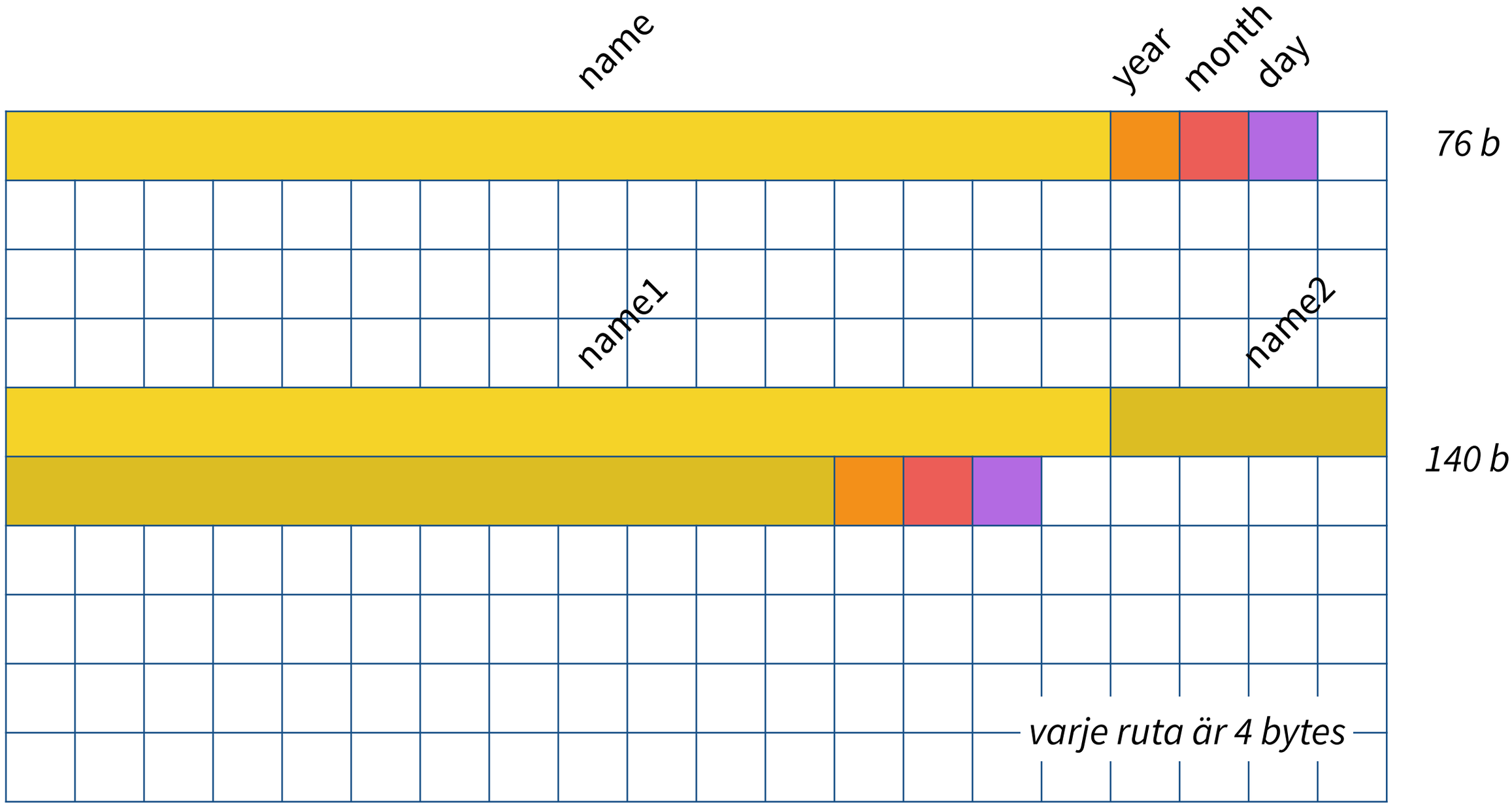
- Ingen pattern matching
- Inget enkelt sätt att konstruera eller dekonstruera
- Definitioner för att skapa ett sammanhängande datablock med olika *namngivna fält* som innehåller *värden av fix storlek*.

Ingen enkel motsvarighet till `Anniversary`

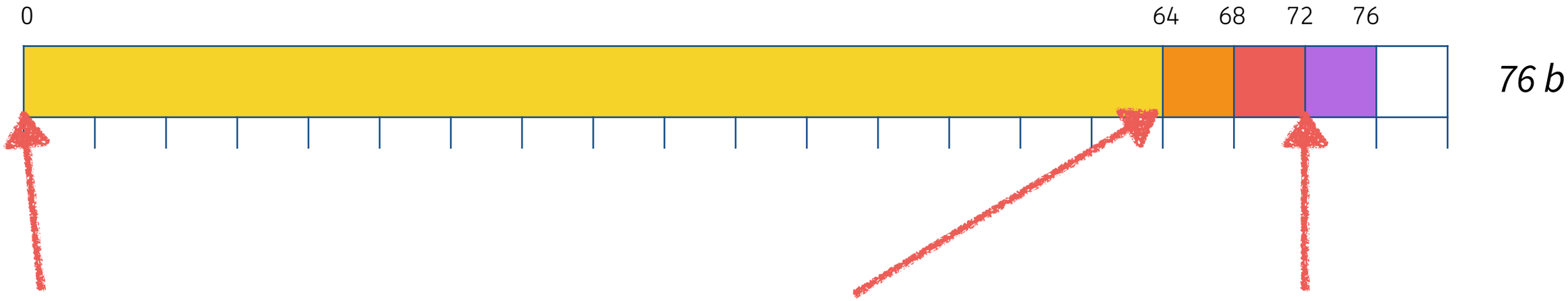
```c
struct birthday
{
  char name[64];
  int year;
  int month;
  int day;
};
```

```c
struct wedding
{
  char spouse_name1[64];
  char spouse_name2[64];
  int year;
  int month;
  int day;
};
```

En strukt är en ”minneskarta” [char=1 byte, int=4 bytes nedan]



En strukt är en ”minneskarta” [char=1 byte, int=4 bytes nedan]




```
&bday.year == &bday + sizeof(char[64])

&bday.month == &bday + sizeof(char[64]) + sizeof(int)
```

Unions — inte riktigt algebraiska datatyper…

- En union tillåter oss att gömma flera typer i en — som *alternativ*

  Storleken (**`sizeof`**) på en union av T_1 och T_2 är *max*(**`sizeof`**(T_1), **`sizeof`**(T_2))

  Metadata för att veta vilken typ ett värde har måste stoppas in för hand

  Jmf. `kind` nedan

```
enum anniversary_kind { WEDDING, BIRTHDAY };

struct anniversary
{
  enum anniversary_kind kind;
  union
  {
    struct birthday b;
    struct wedding w;
  };
};
```

Unions — inte riktigt algebraiska datatyper…

- En union tillåter oss att gömma flera typer i en — som *alternativ*

  Storleken (**`sizeof`**) på en union av T_1 och T_2 är *max*(**`sizeof`**(T_1), **`sizeof`**(T_2))

  Metadata för att veta vilken typ ett värde har måste stoppas in för hand

  Jmf. `kind` nedan

```
void use_anniversary(struct anniversary a)
{
  if (a.kind == BIRTHDAY)
    {
      a.b.name = ...
    }
  else
    {
      a.w.spouse_name1 = ...
    }
}
```

Att deklarera en variabel av strukt-typ

- Den nedre varianten är att föredra

  Notera att den **nollställer allt** som inte tilldelas

```
// #1
struct wedding w; // uninitialised wedding
w.year = 1972;
// etc.

// #2
struct birthday b =
  (struct birthday) { .year = 1972, .month = 5, .day = 21 };
```

(union fungerar likadant)

Att läsa eller tilldela poster i en strukt

- Den nedre varianten är att föredra

Notera att den **nollställer allt** som inte tilldelas

```
struct wedding w;
// uppdaterar day-posten
w.day = 30;

// läser (och skriver ur) spouse_name2-posten
puts(w.spouse_name2);


struct anniversary a = ...;
puts(a.w.spouse_name2);
```

”navigera” genom nästlade struktar/unioner

värdesemantik!

```c
#include <stdio.h>

struct point { int x, y; };

void print_point(struct point p)
{
  printf("(%d,%d)\n", p.x, p.y);
}

void move_point(struct point p, int dx, int dy)
{
  p.x += dx;
  p.y += dy;
}

int main(void) {
  struct point p = { 100, 50 }; // Valitt, men felbenäget!
  print_point(p);
  move_point(p, 10, 10);
  print_point(p);
  return 0;
}
```

pekarsemantik!

```c
#include <stdio.h>

struct point { int x, y; };

void print_point(struct point p)
{
  printf("(%d,%d)\n", p.x, p.y);
}

void move_point(struct point *p, int dx, int dy)
{
  p->x += dx;
  p->y += dy;
}

int main(void) {
  struct point p = { 100, 50 }; // Valitt, men felbenäget!
  print_point(p);
  move_point(&p, 10, 10);
  print_point(p);
  return 0;
}
```

Pekarsemantik vs. värdesemantik

- Värdesemantik

  Kopierar data — problem med identitet, men inga problem med förändringar

  Kostar att kopiera data

- Pekarsemantik

  Delar data — identitet bibehålls, men/och förändringar får ”globalt genomslag”

  Kan vara dyrt att läsa data som är långt borta (eller negativ cache coherence)

- Vad är rätt i vilket program/sammanhang?

Typedef

Vi kommer att använda `typedef` *från och med nu*

```c
#include <stdio.h>

typedef struct point point_t;

struct point { int x, y; };

void print_point(point_t p)
{
  printf("(%d,%d)\n", p.x, p.y);
}

void move_point(point_t *p, int dx, int dy)
{
  p->x += dx;
  p->y += dy;
}
```

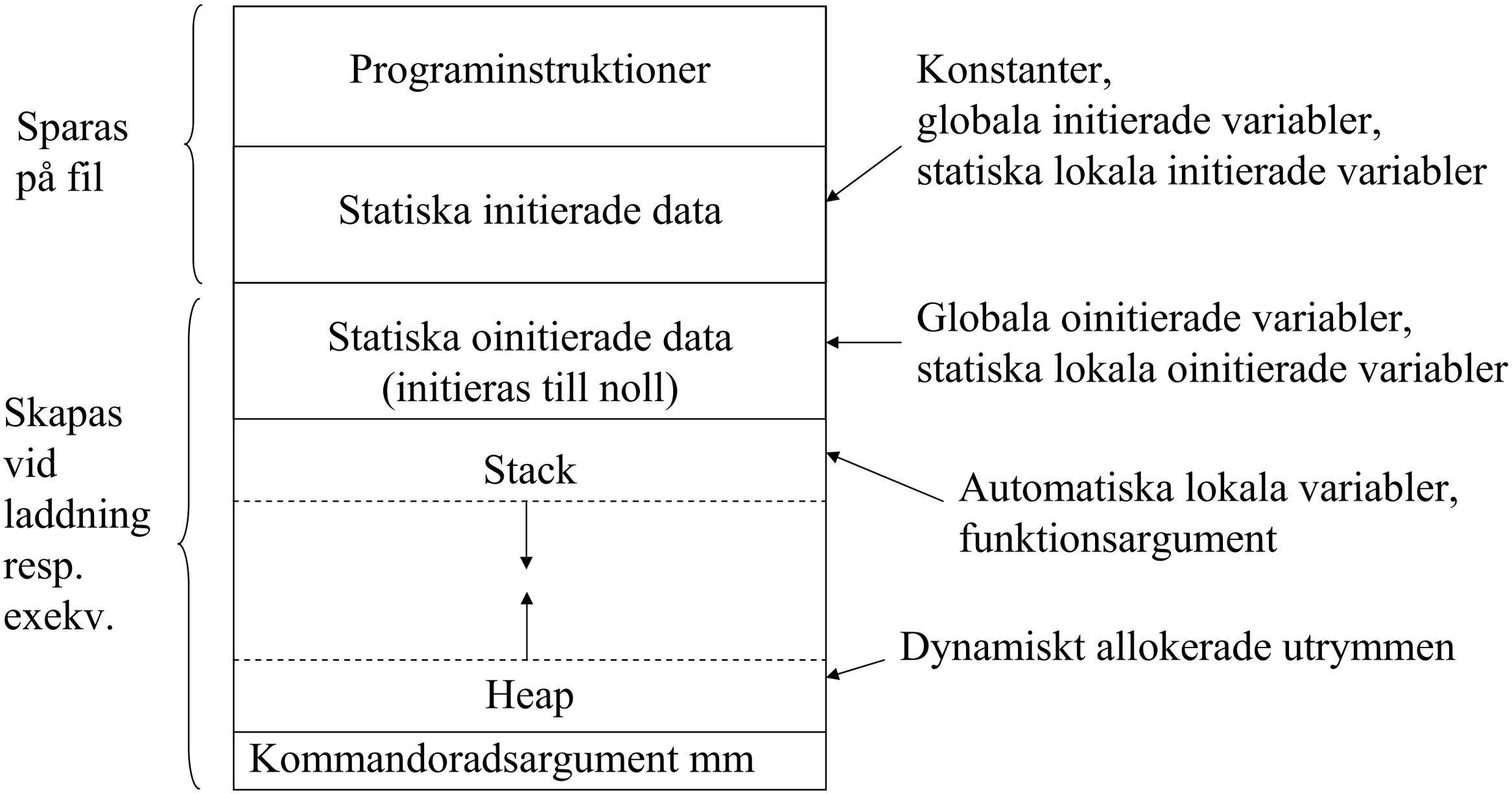
```c
int main(void) {
  point_t p = (point_t) { .x = 100, .y = 50 };
  print_point(p);
  move_point(p, 10, 10);
  print_point(p);
  return 0;
}
```

Föreläsning 1–3

Tobias Wrigstad

Stack & heap, manuell minneshantering, pekare

Var lagras data?



Stacken

Läs Addendum om stack och heap

Ett enkelt stackexempel

Syftet med det här lilla häftet är att driva hem några poänger kring skillnaderna mellan stack och *heap*, adressutagningsoperatorn & och avreferensingsoperatorn *.

Vi börjar med ett enkelt kodexempel:

```
int x = 42;
int *y = &x;
int *z = NULL;
z = y;
(*z)++;
y++;
```

Vilka värden har variablerna x, y och z? Svaret är att värdet på x är 43 men att värdena på y och z är okända[1].

Men låt oss börja från början. För enkelhetens skull kan vi tänka oss att ett C-program har tillgång till två sorters minne: *stackminne* och *heapminne*. Stacken är det minnesutrymme som används för att lagra värdena i lokala variabler i funktioner, så kallade automatiska variabler. Betrakta följande funktion:

```
void stupid(int value) {
  if (value) {
    int smaller = value - 1;
    stupid(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 `smaller`.[2]

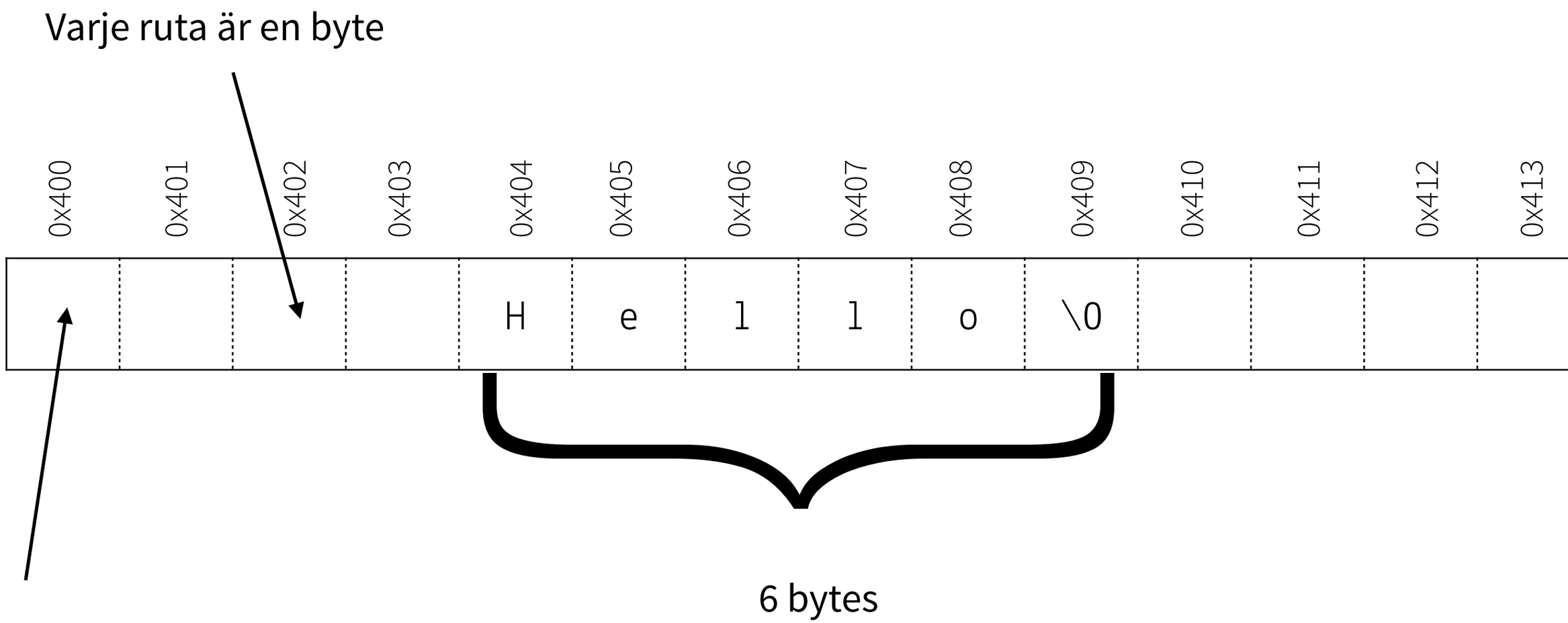
Funktionen `stupid` är rekursiv; ett anrop `stupid(3)` kommer att leda till att funktionen anropar sig själv ytterligare tre gånger; varje funktion har en egen lokal variabel `value` vars värde är ett mindre än den anropande funktionens `value`-värde. Varje gång `stupid` anropas behövs ytterligare 8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

[1] Observera att vet vi att om värdet på z är heltalet n, så är värdet på y n + 4, givet att storleken på en pekare är 4 bytes.

[2] Det är en liten förenkling – vissa andra data kan behövas, och smarta kompilatorer gör optimeringar som innebär bort variabler som inte behövs, som t.ex. `smaller` här. Vi bortser från sådd.

https://github.com/IOOPM-UU/ioopm17/blob/master/extramaterial/C-addendum-om-stack-och-heap.pdf

C:s minnesmodell



Ett enkelt C-program & dess minnesanvändning

```c
#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

1 byte

? bytes

1 byte

4 bytes

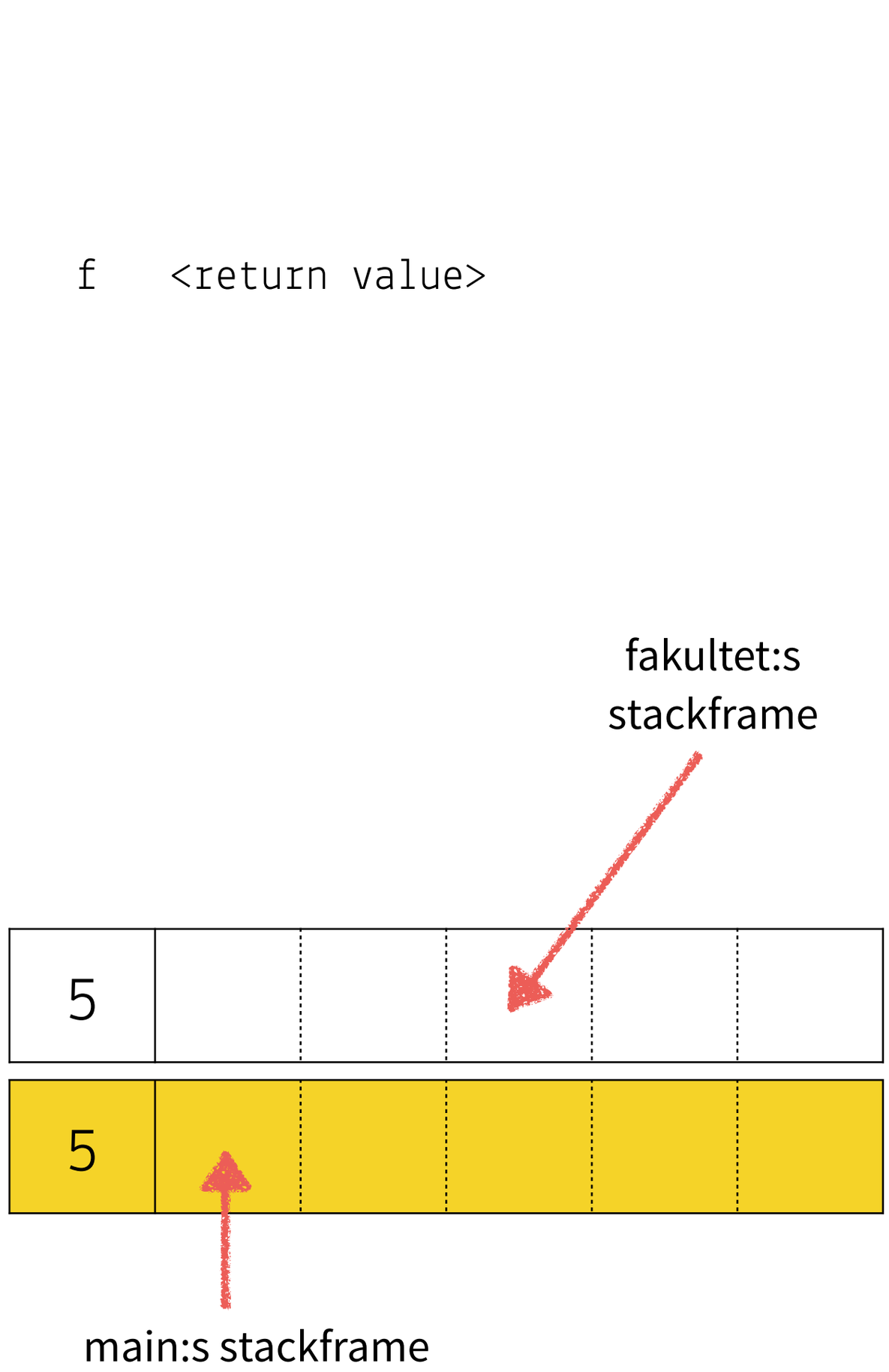
?? bytes

+binären, etc.

```c
#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;
}
```

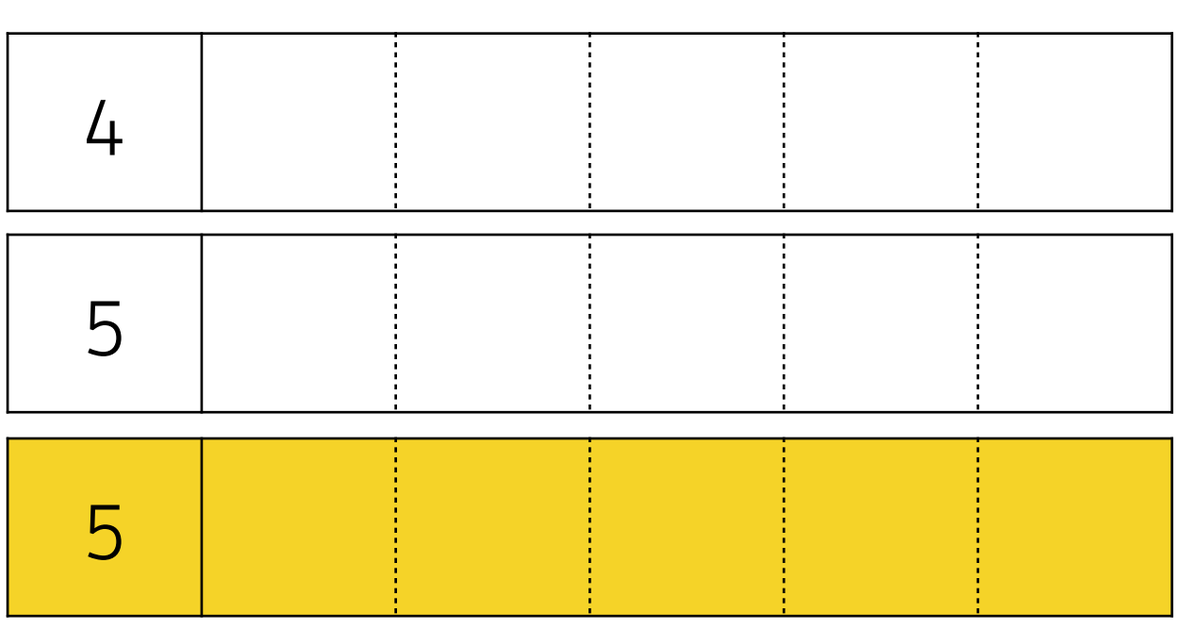
```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

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>
```



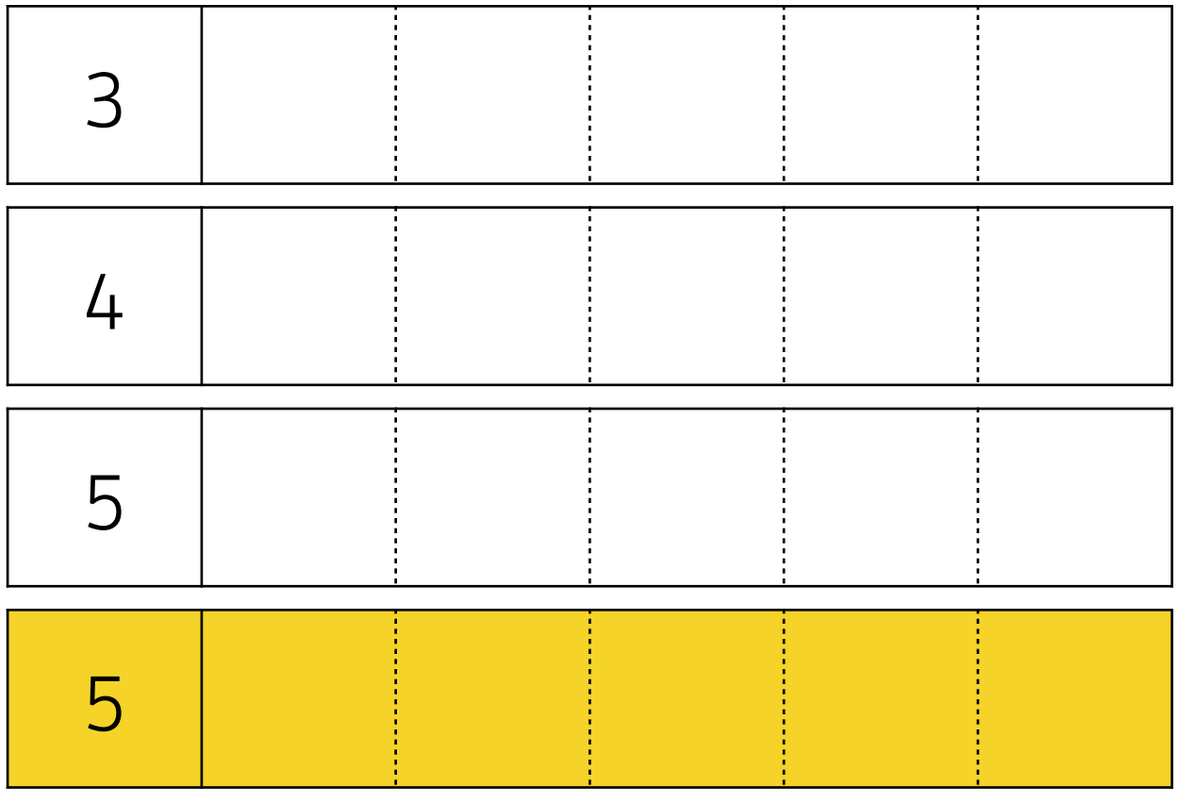
```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

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>
```

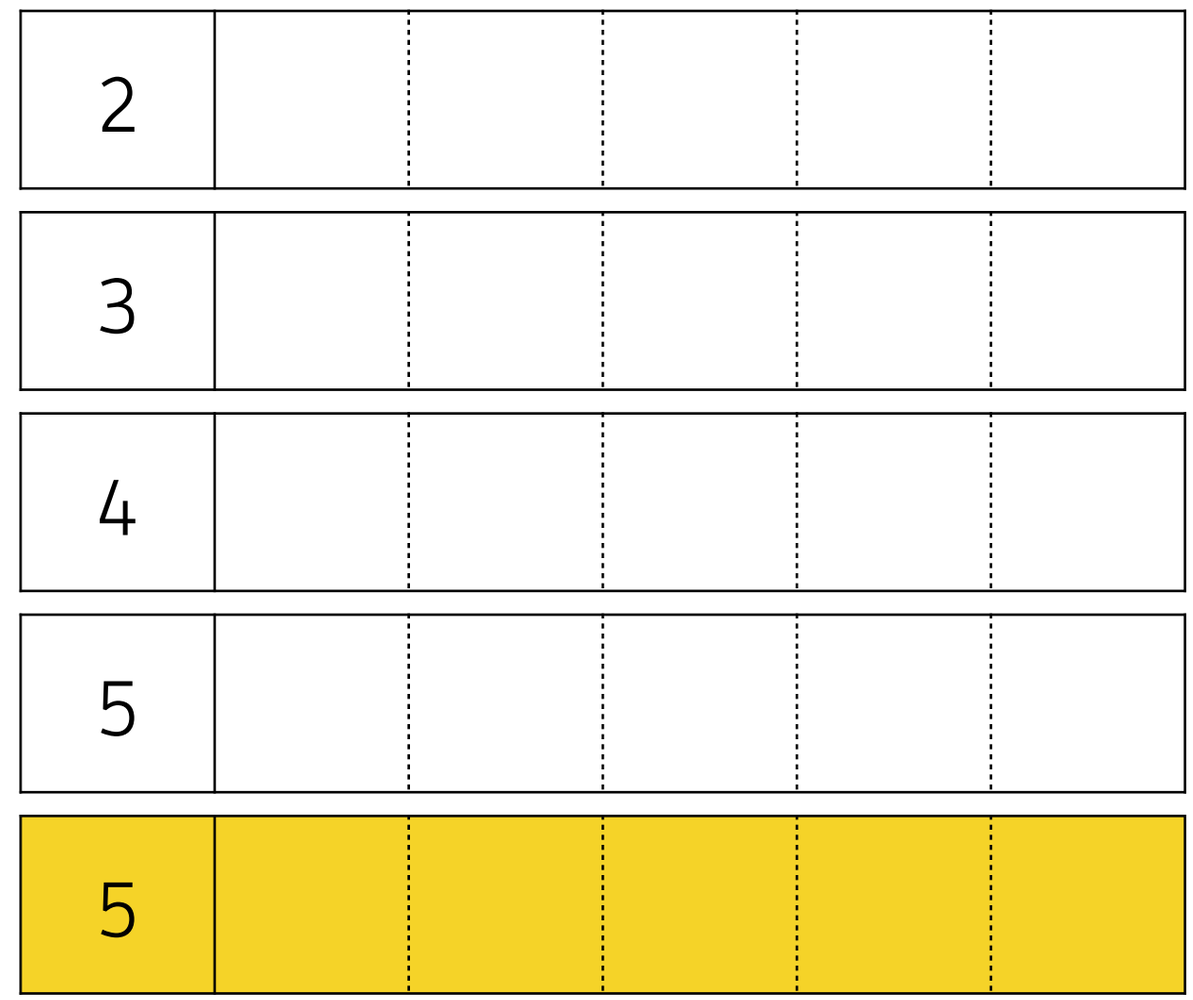


```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

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;
}
```

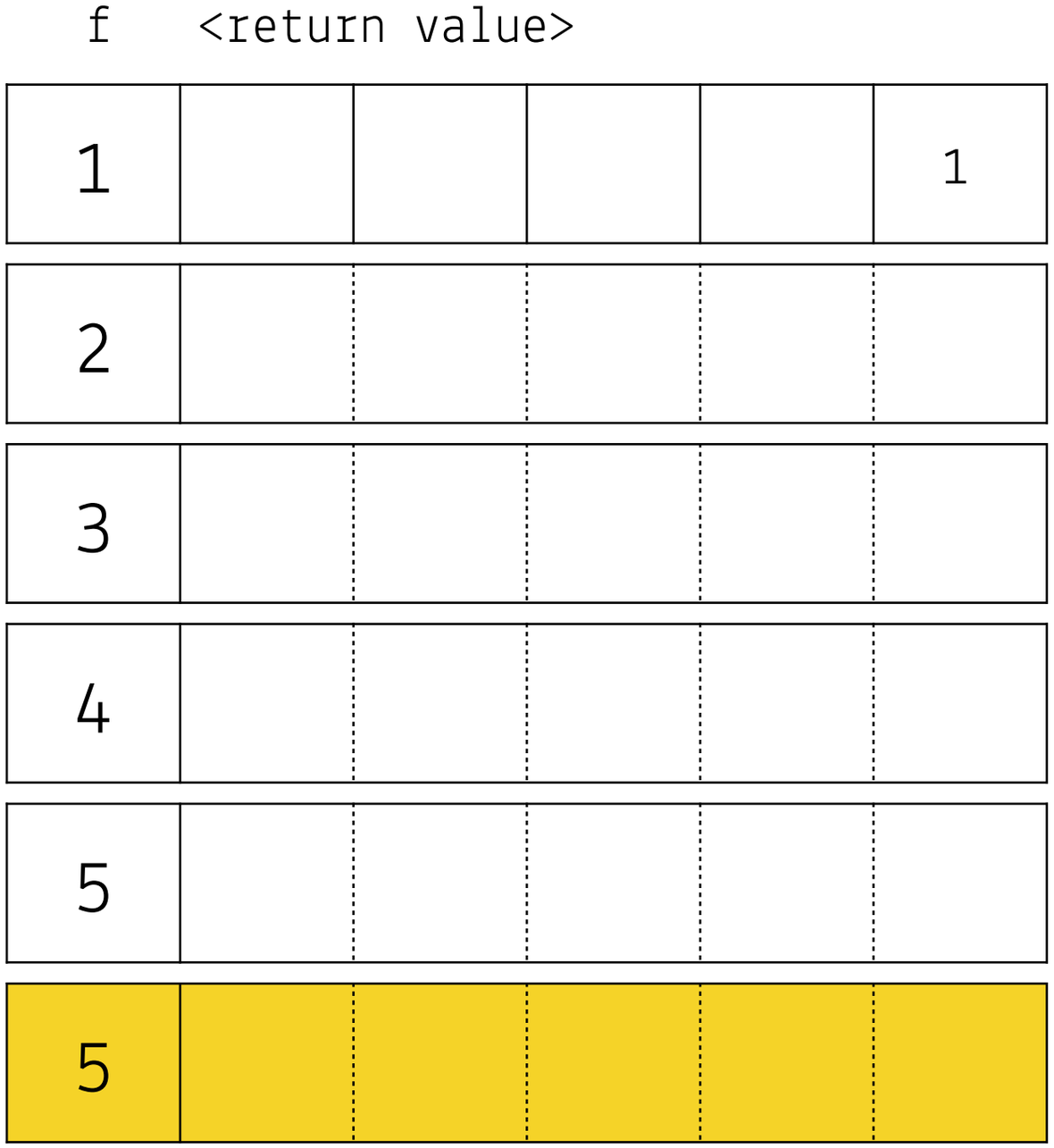
f <return value>



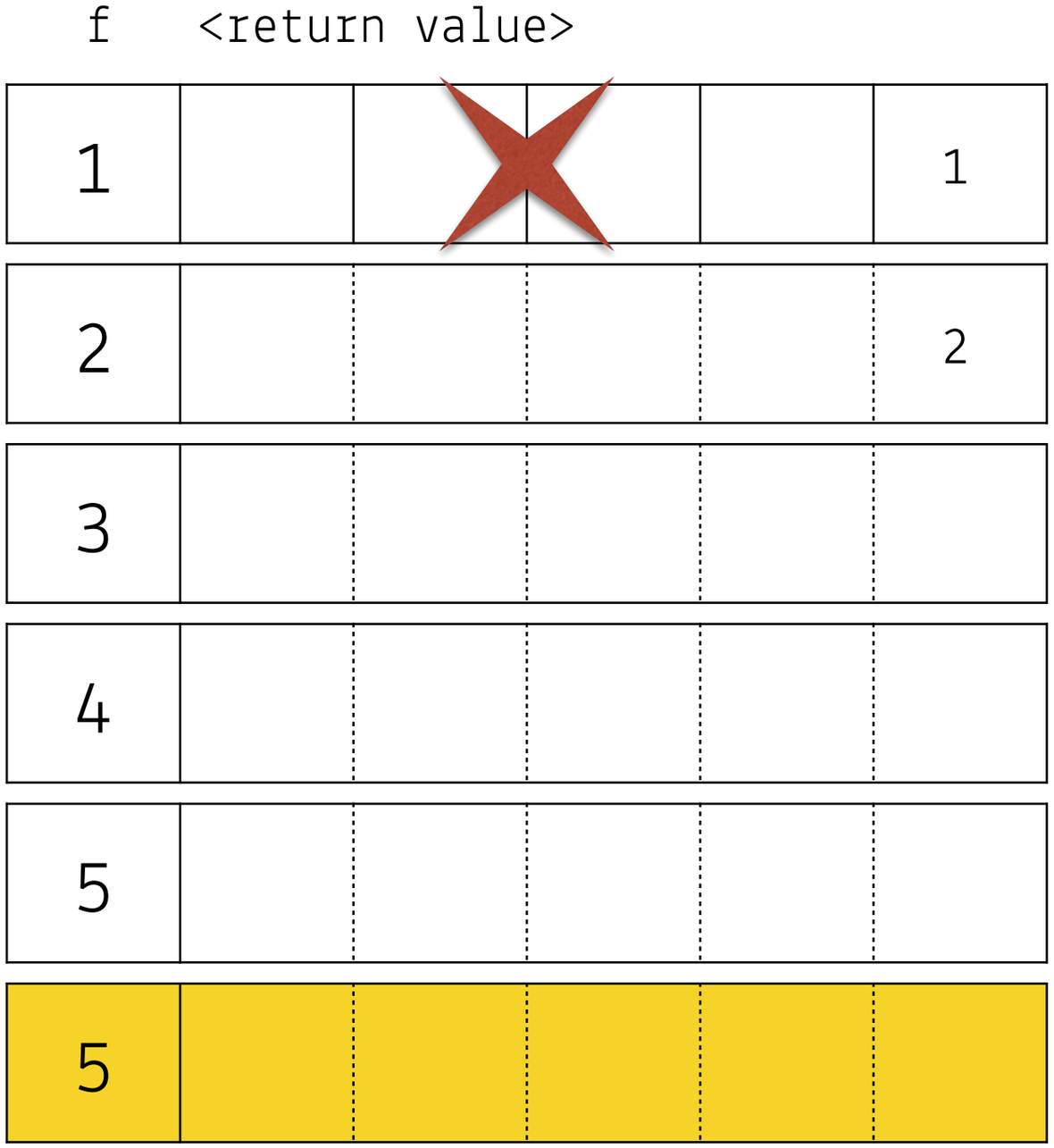
```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

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;
}
```



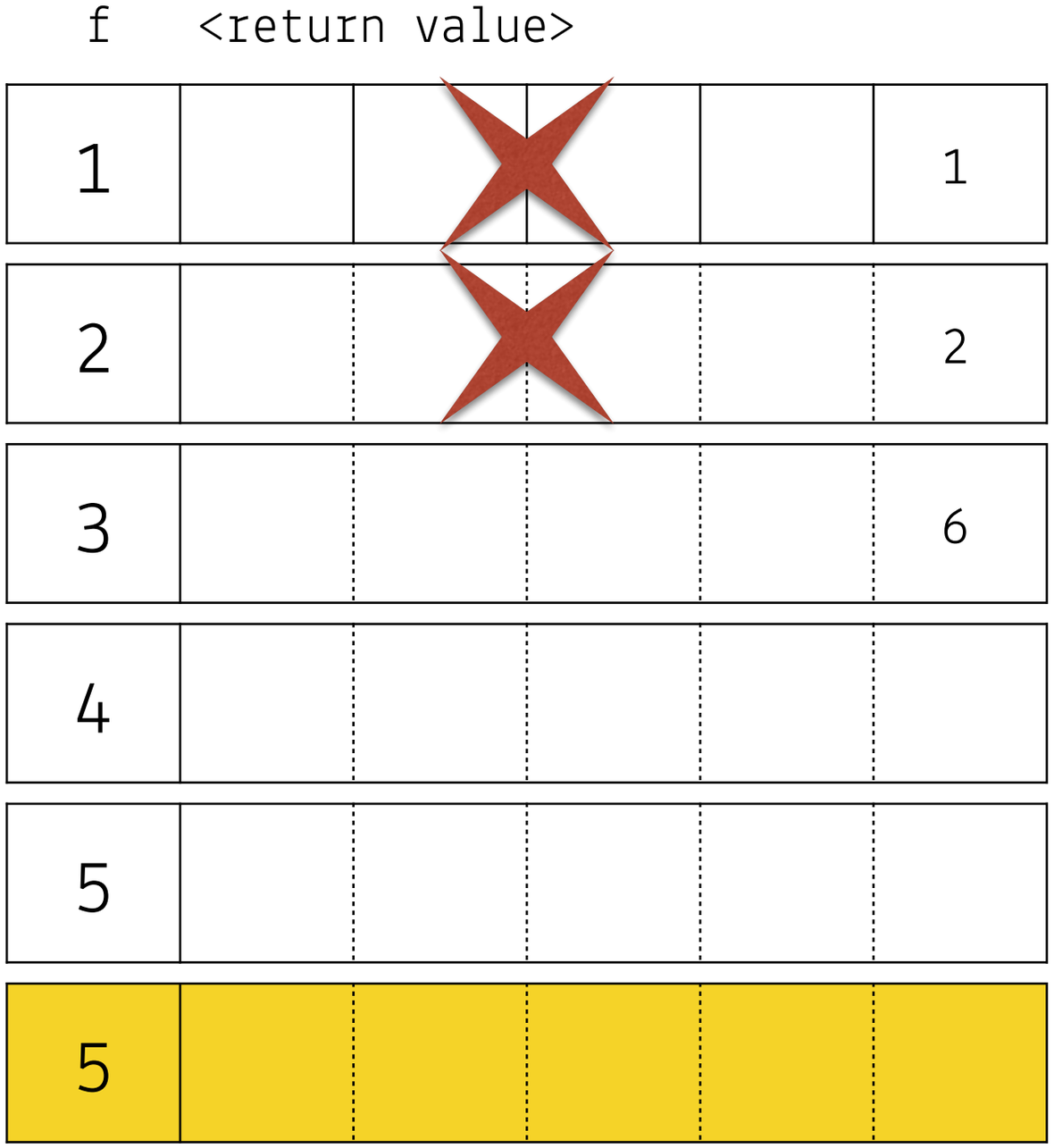
```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

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;
}
```



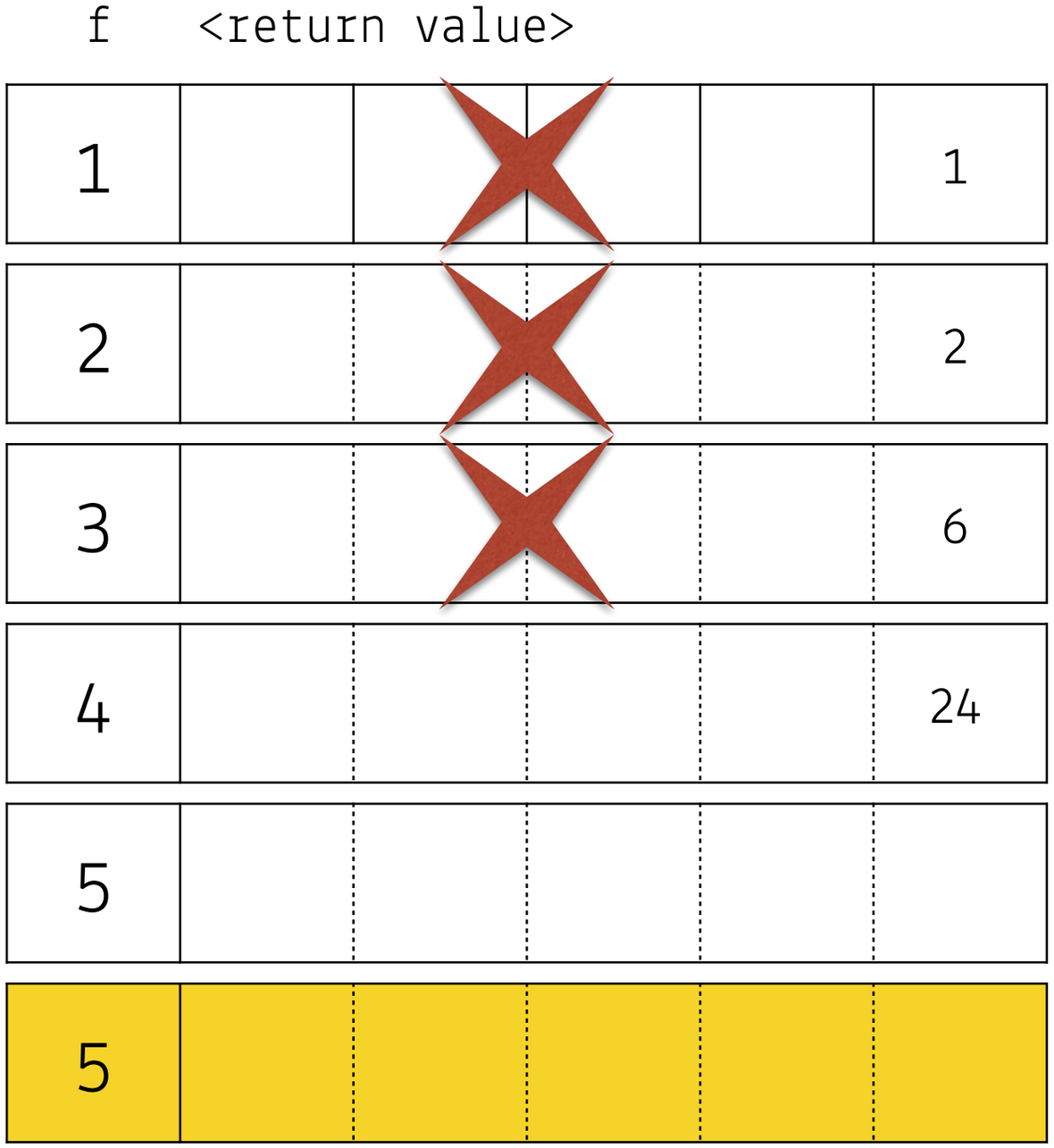
```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

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;
}
```



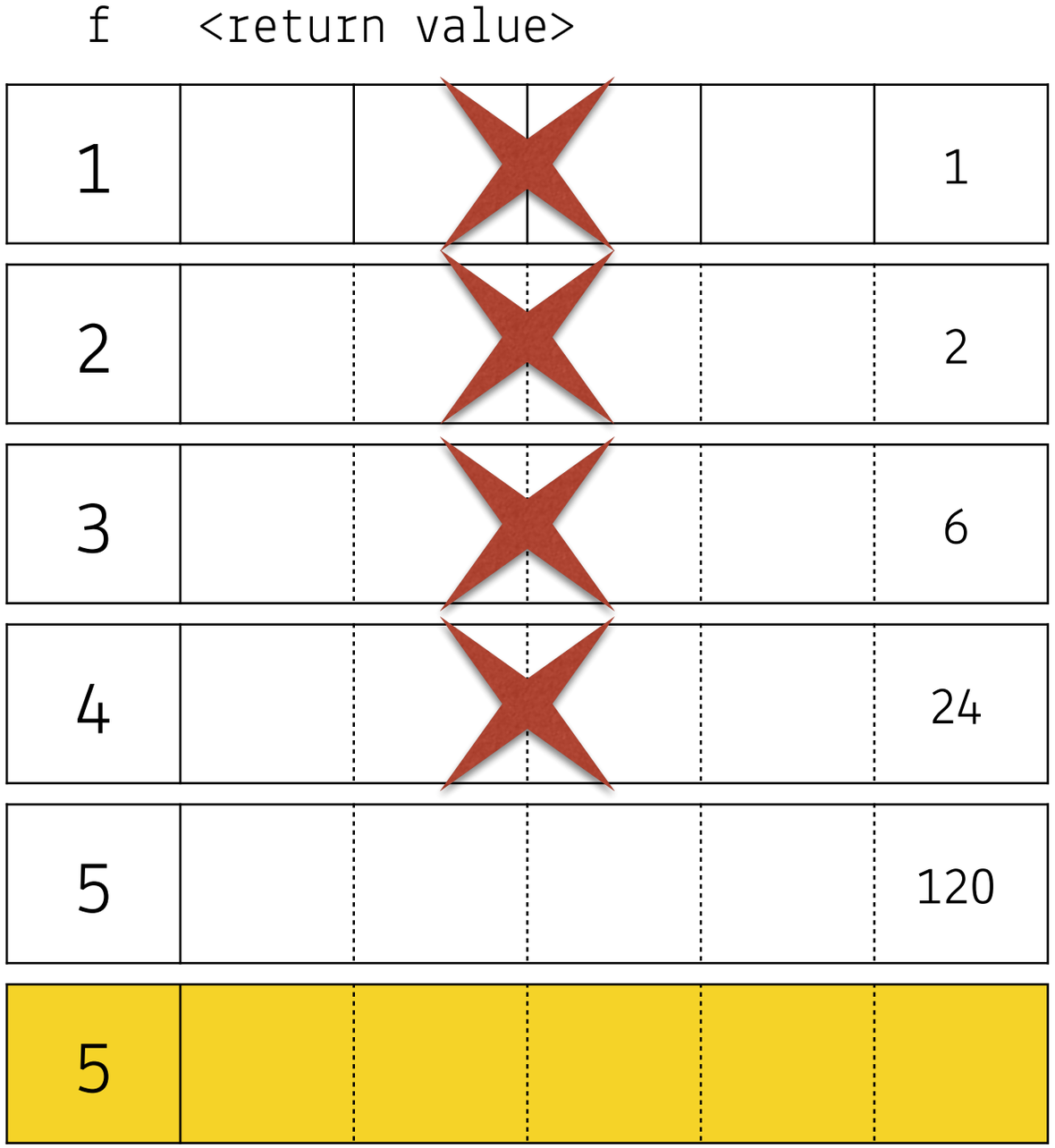
```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

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;
}
```



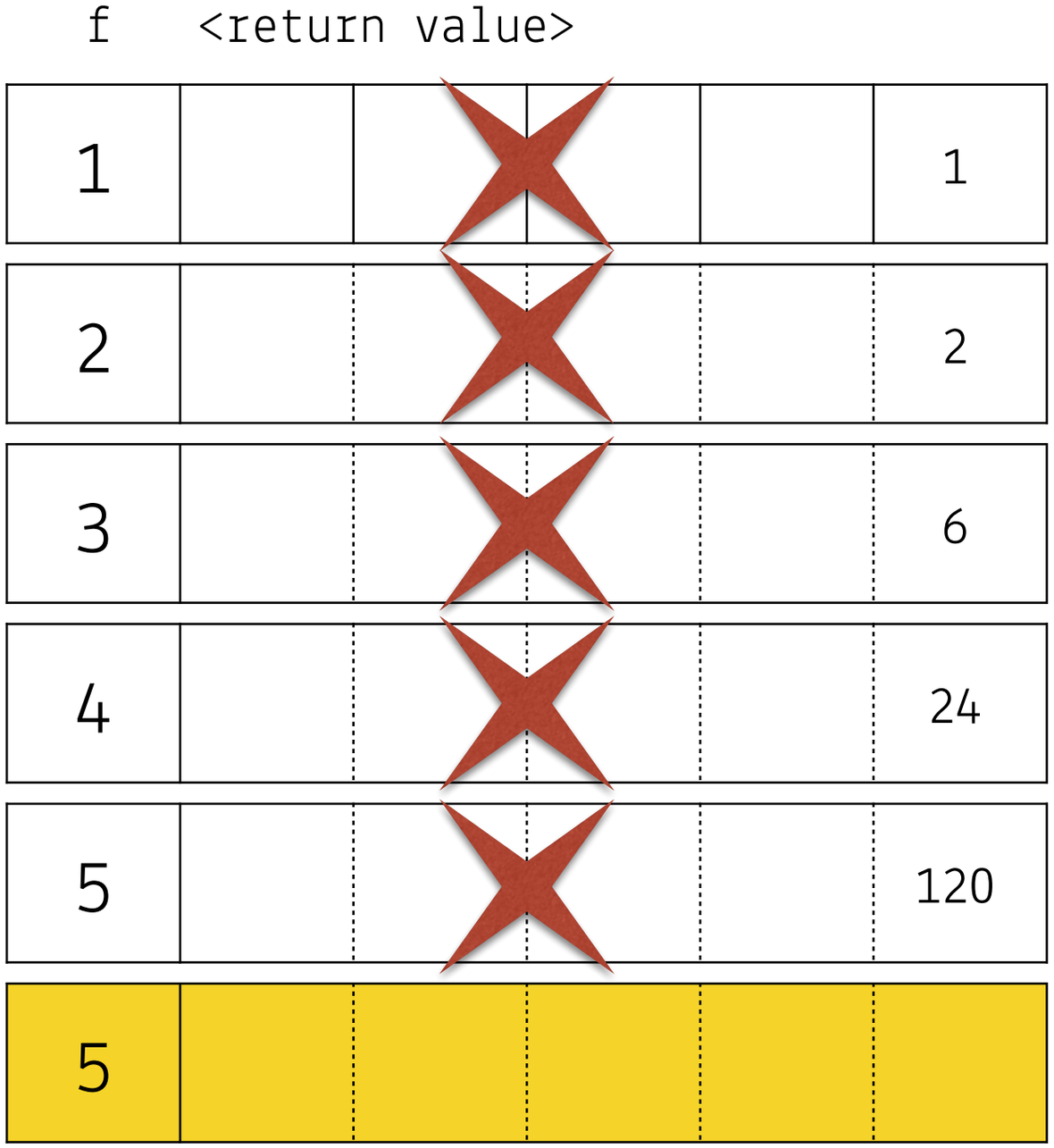
```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

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;
}
```



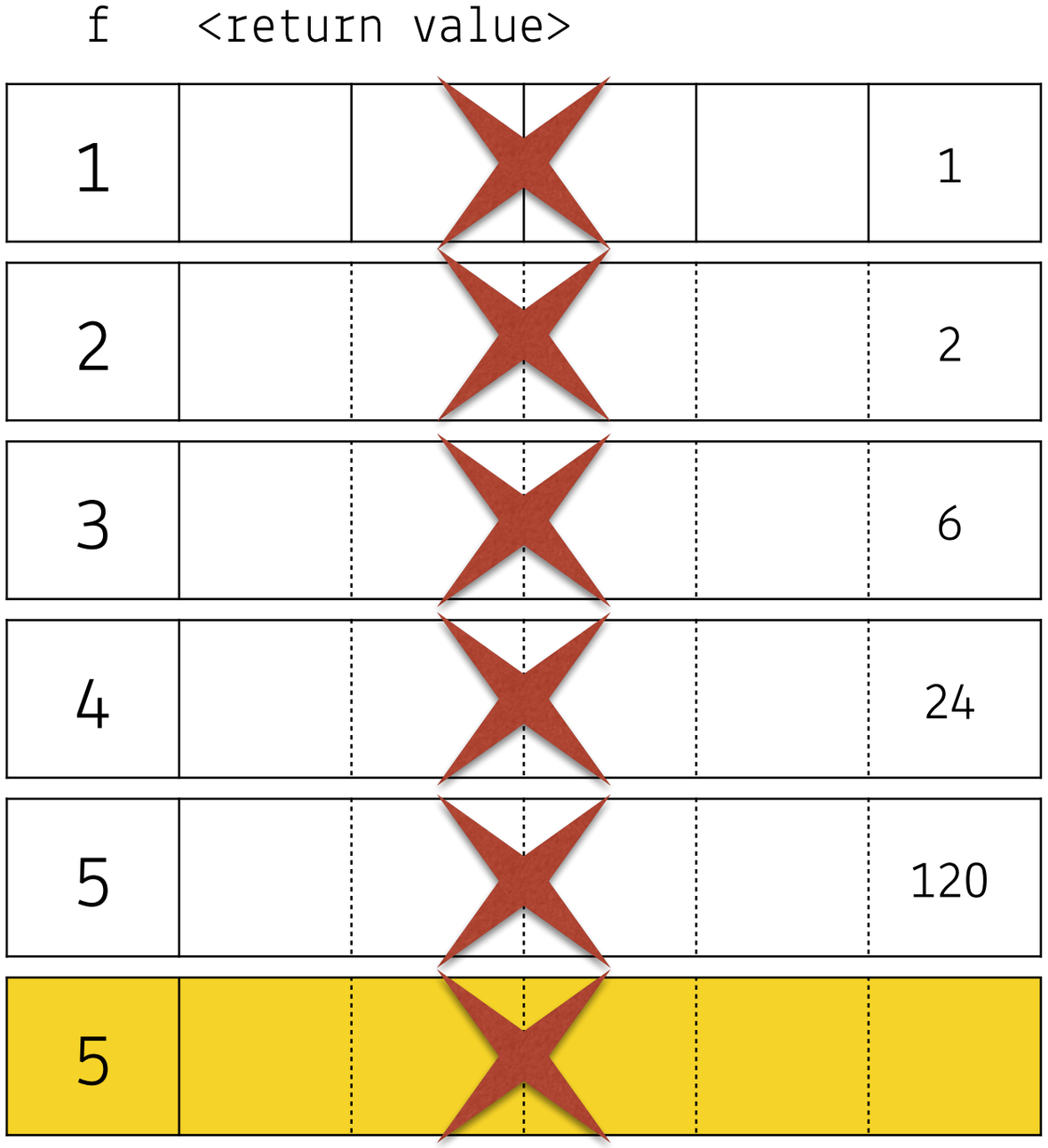
```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

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;
}
```



```c
#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;
}
```

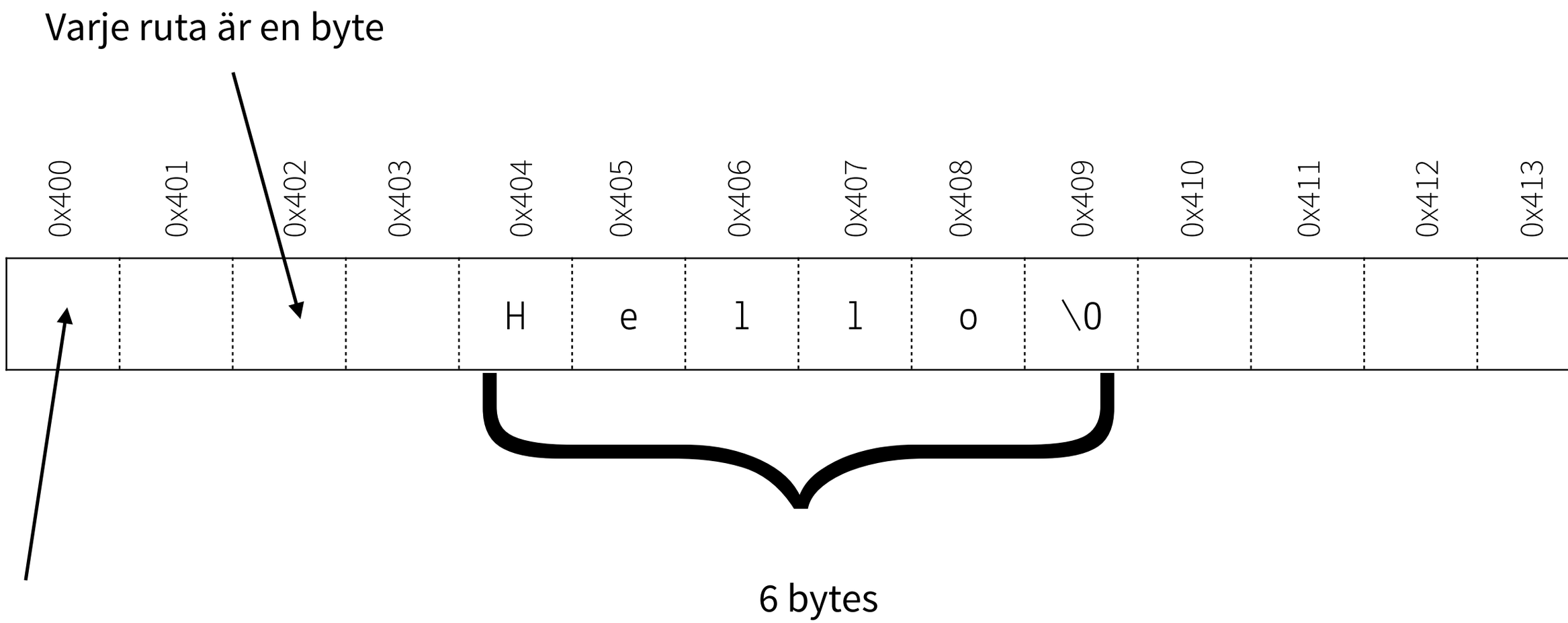


```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

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;
}
```



”Pekare”

C:s minnesmodell



Pekare

- Variabler som innehåller adresser till platser i minnet
- Två operationer:

  Peka om variabeln

  Avreferera pekaren för att läsa/skriva minnesplatsen

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

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


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

```
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;        // uppdatera pekaren p
```

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


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

adresstagnings-
operatorn

```
p = &x;        // uppdatera pekaren p
```

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


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

adresstagnings-
operatorn

```
p = &x;        // uppdatera pekaren p


               // uppdatera minnesplatsen som p
*p = x;        // pekar på
```

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


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

adresstagnings-
operatorn

```
p = &x;        // uppdatera pekaren p
```

```
*p = x;        // uppdatera minnesplatsen som p
               // pekar på
```

avrefererings-
operatorn

”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”

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

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

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

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

NULL POINTER DEREFERENCE

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

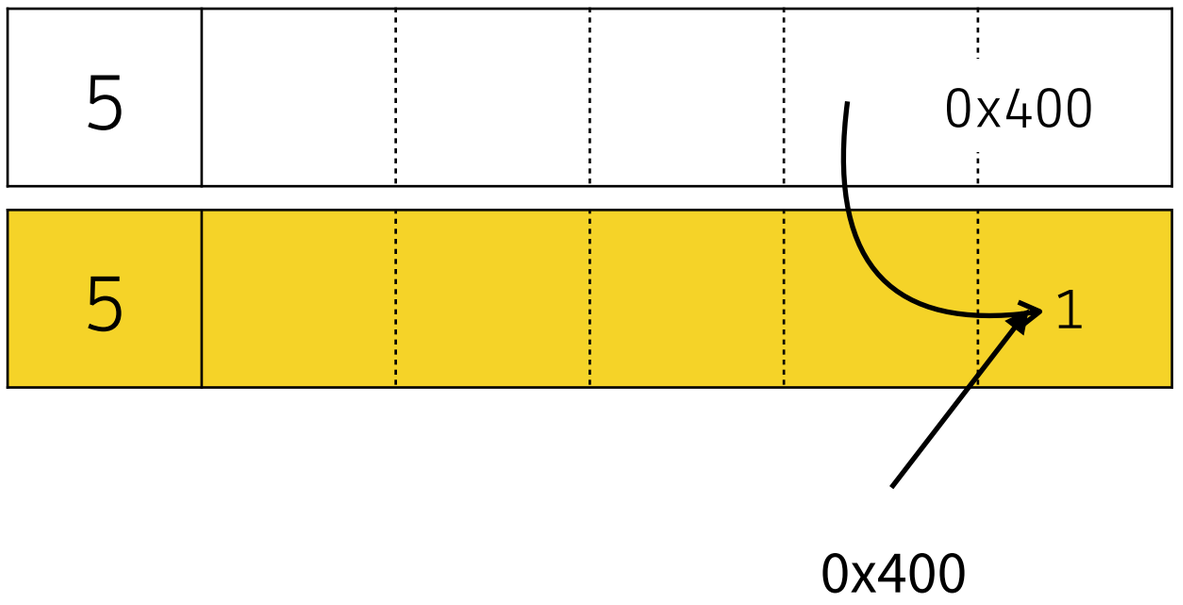
`*var` — följ en pekare till en plats i minnet som håller data

`NULL` — en pekare som inte pekar på någonting

Återbesök av stackexemplet med pekare

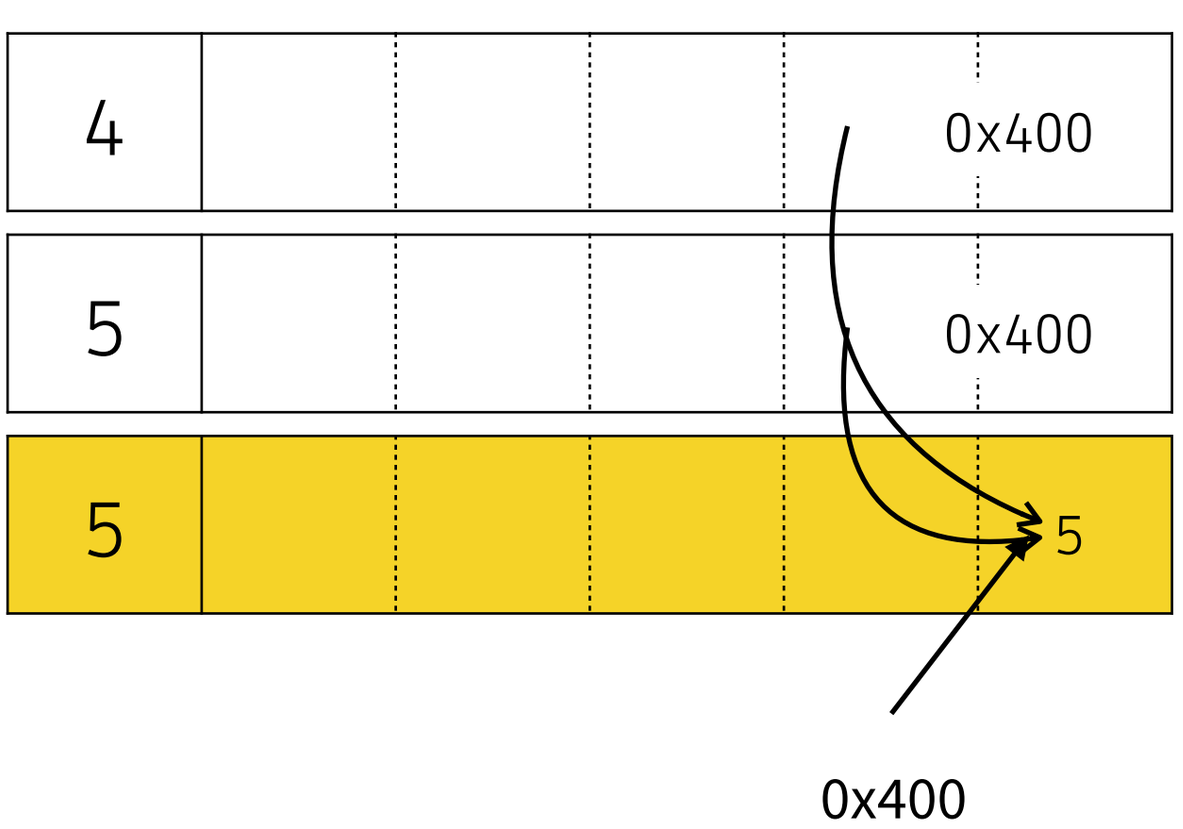
```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

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>



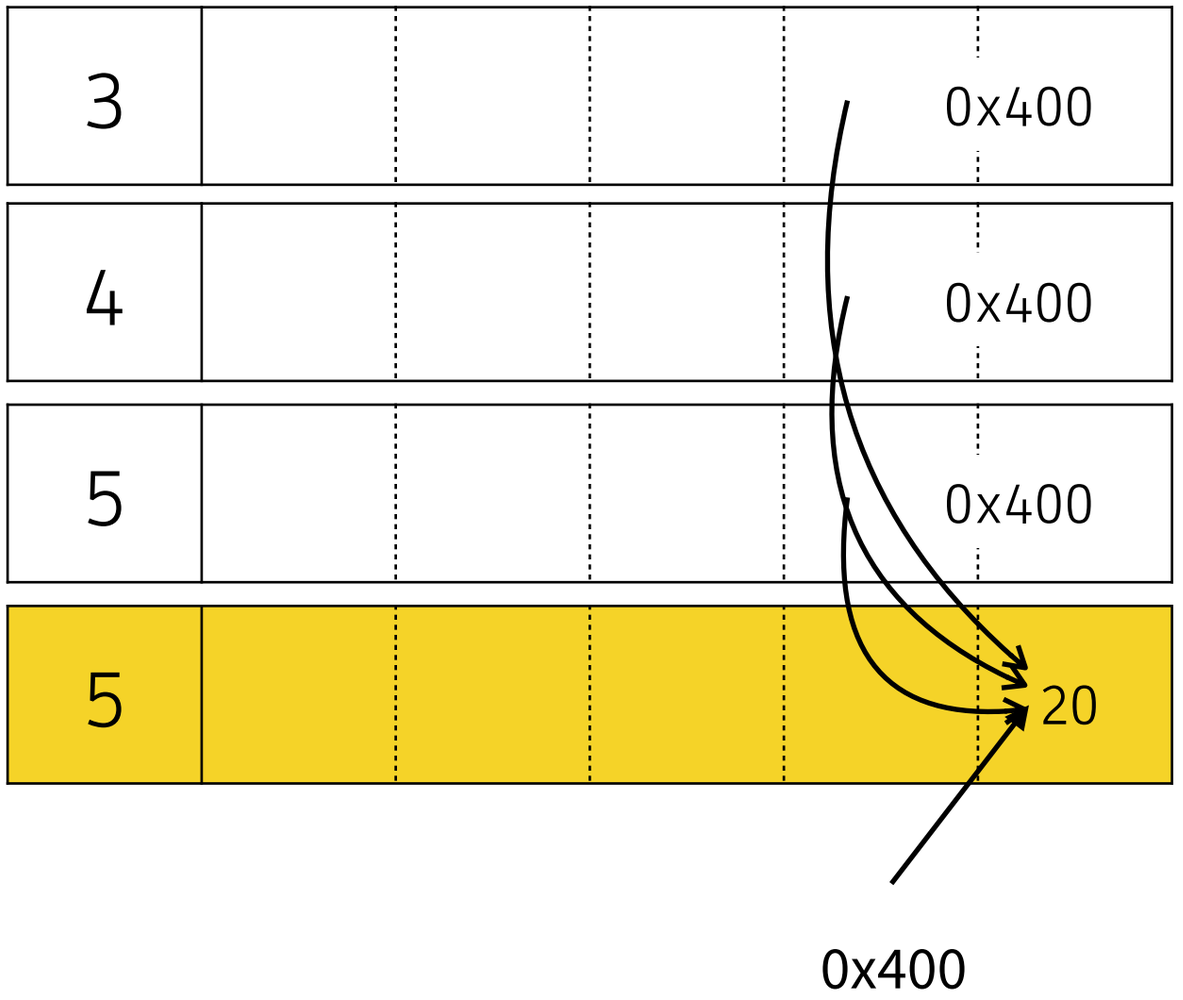
```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

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>

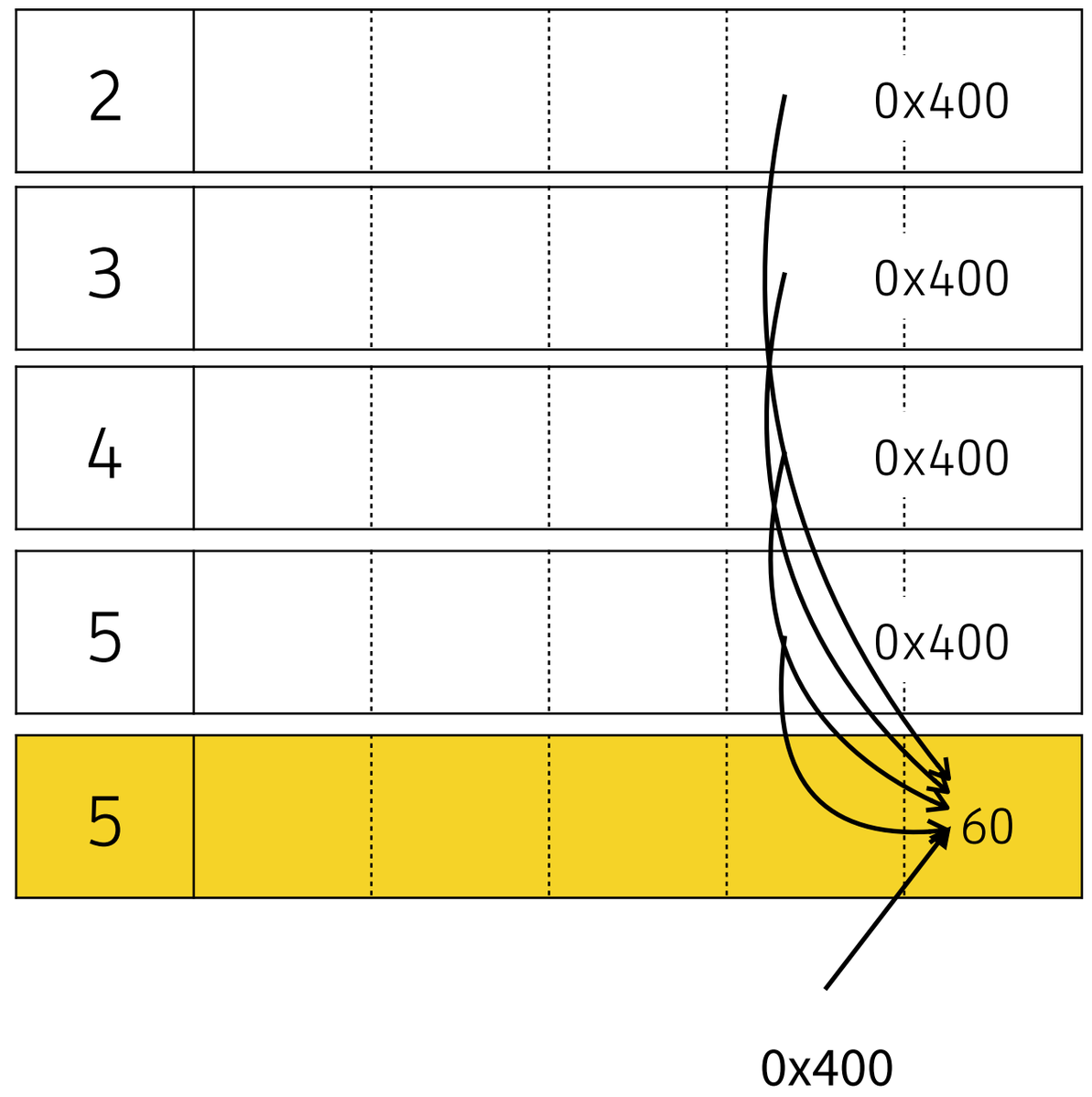


```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

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;
}
```

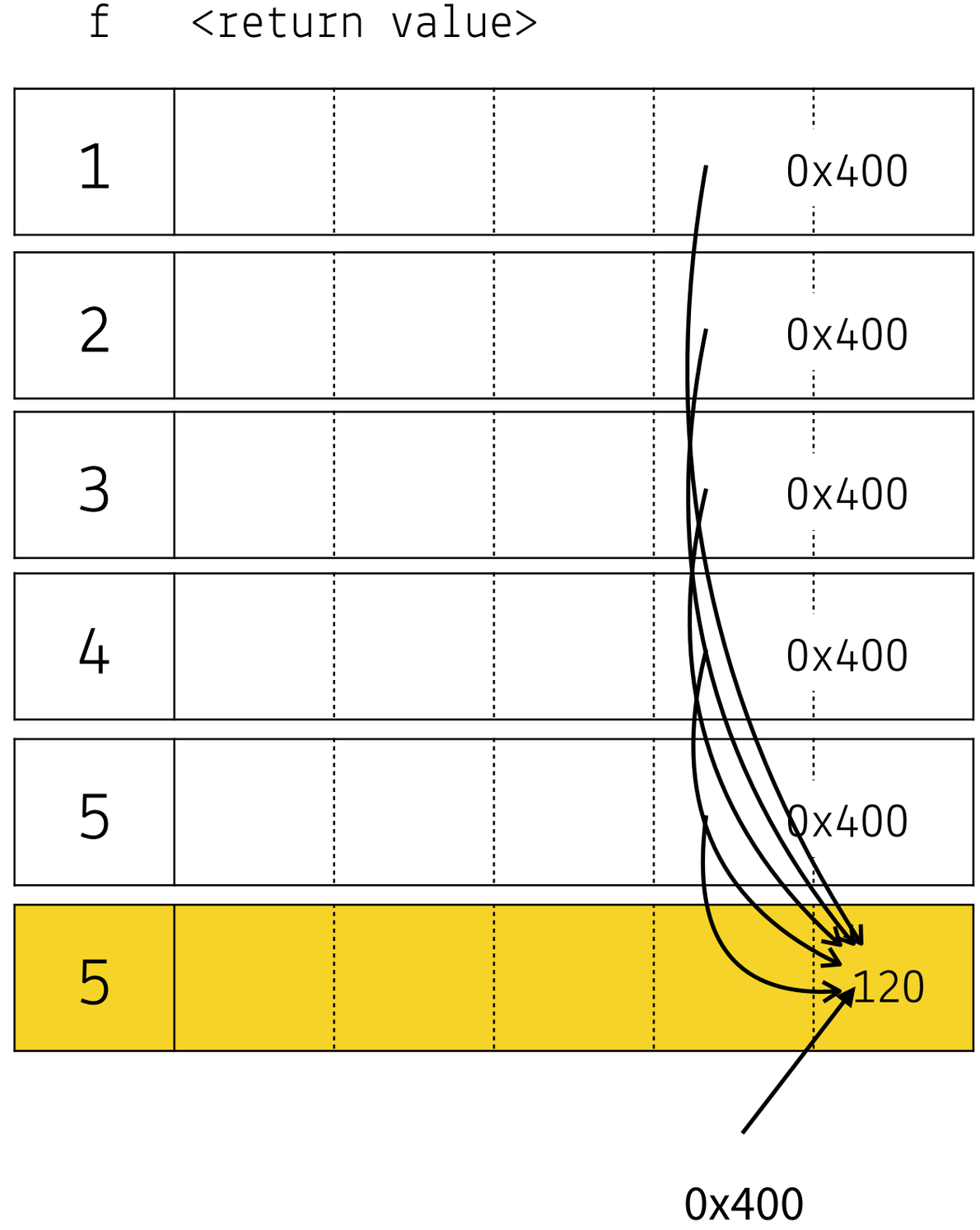
f <return value>



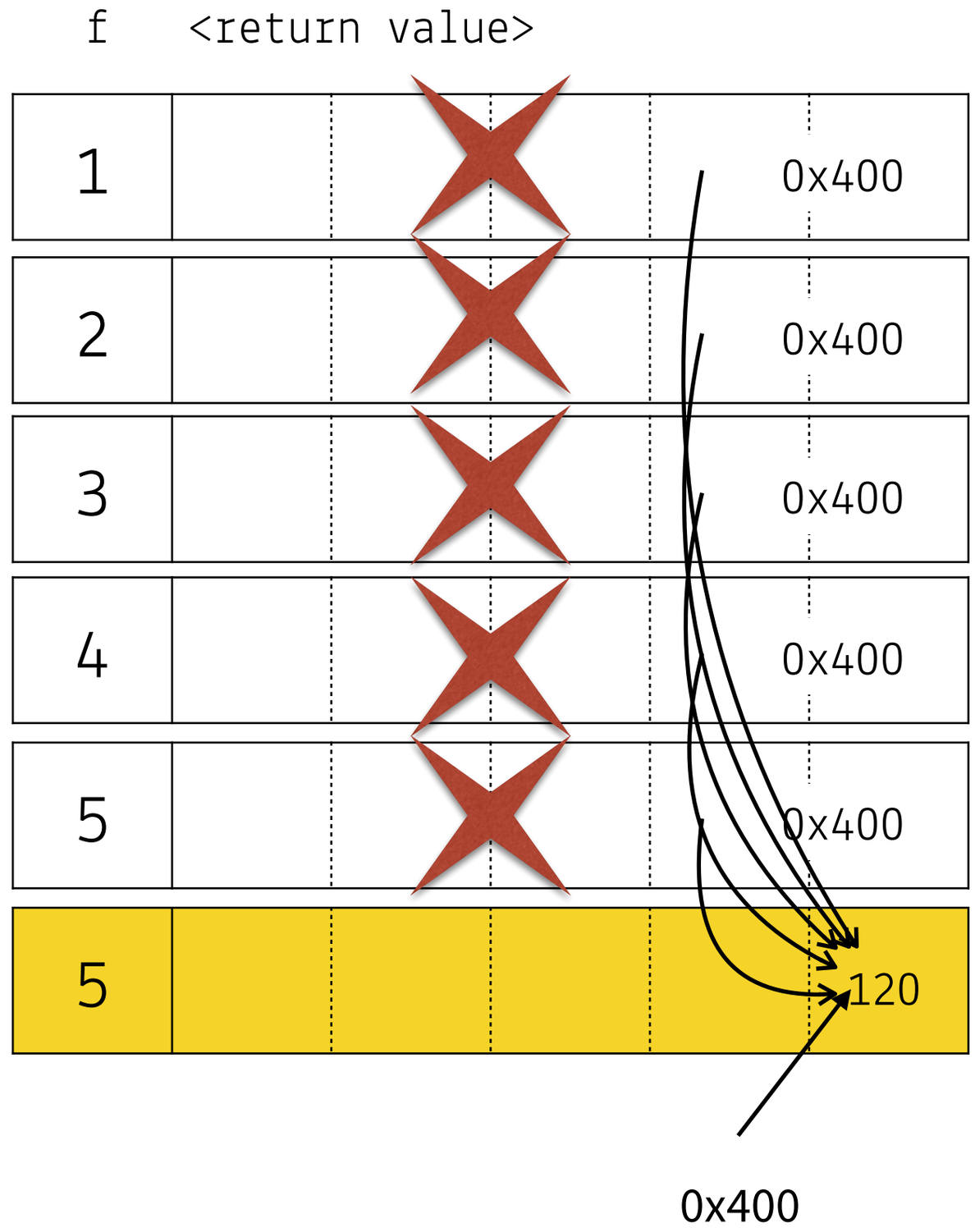
```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

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;
}
```



```c
#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;
}
```



```c
#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;
}
```



```c
#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)

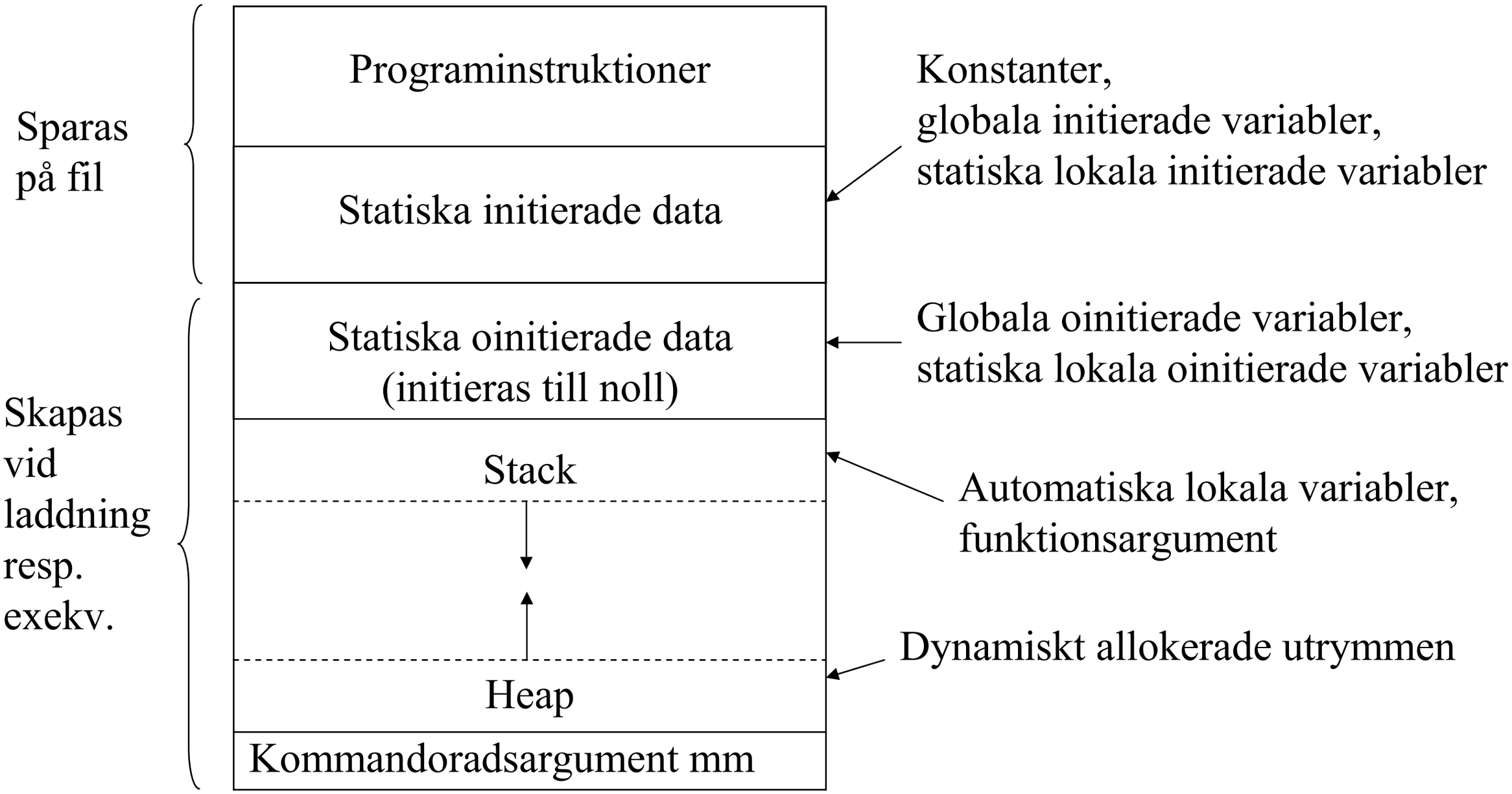
  Varje funktionsanrop ger upphov till en ny ”stack frame”
  (not. parameteröverföring sker i regel mha register)

  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?



Heapen

Läs Addendum om stack och heap

Ett enkelt stackexempel

Svenet mxo det här lilla häftet är att driva hem några poänger kring skillnaderna mellan stack och heap, adressutagningsoperatorn & och avreferensoperatorn *.

Vi börjar med ett enkelt kodexempel:

```
int x = 42;
int *y = &x;
int **z = NULL;
z = y;
(*z)++;
y++;
```

Vilka värden har variablerna x, y och z? Svaret är att värdet på x är 43 men att värdena på y och z är okända[1].

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

```
void stupid(int value) {
  if (value) {
    int smaller = value - 1;
    stupid(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 `smaller`.[2]

Funktionen `stupid` är rekursiv; ett anrop `stupid(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 `stupid` anropas behövs ytterligare 8 bytes för att hålla värdena i dess `value` och `smaller`-variabler. Vi säger att varje funktionsanrop leder till att ny *stack frame* pushas på stacken som innehåller det minnesutrymmen

[1] Däremot så vet vi att om värdet på z är heltalet n, så är värdet på y n + 4, givet att storleken på en pekare är 4 bytes.

[2] Det är en liten förenkling – vissa andra data kan behövas, och smarta kompilatorer gör optimeringar som undlar livet variabler som inte behövs, som t.ex. `smaller` här. Vi bortser från sådant.

https://github.com/IOOPM-UU/ioopm15/blob/master/extramaterial/C-addendum-om-stack-och-heap.pdf

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;
}
```

resultatet måste frigöras — någonstans! (Var och när?)

Dynamiska strukturer

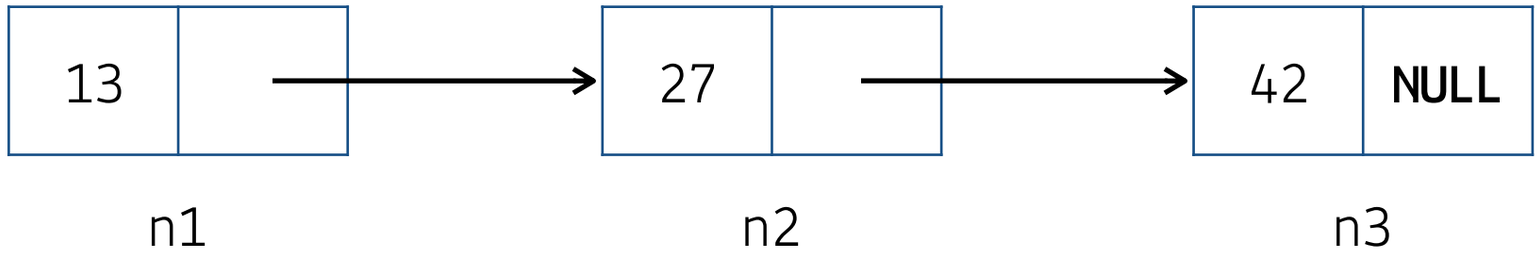
- Heapen används för att lagra stora data och data vars storlek växer dynamiskt, t.ex.

  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!

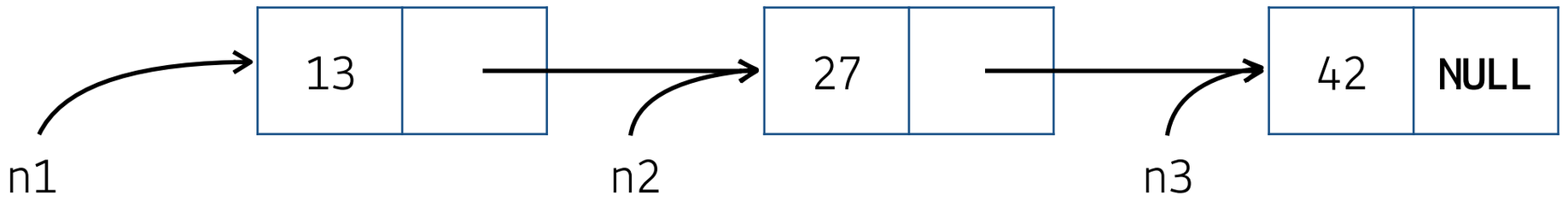
Pekare och länkade strukturer [På stacken]




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

```c
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 };
```

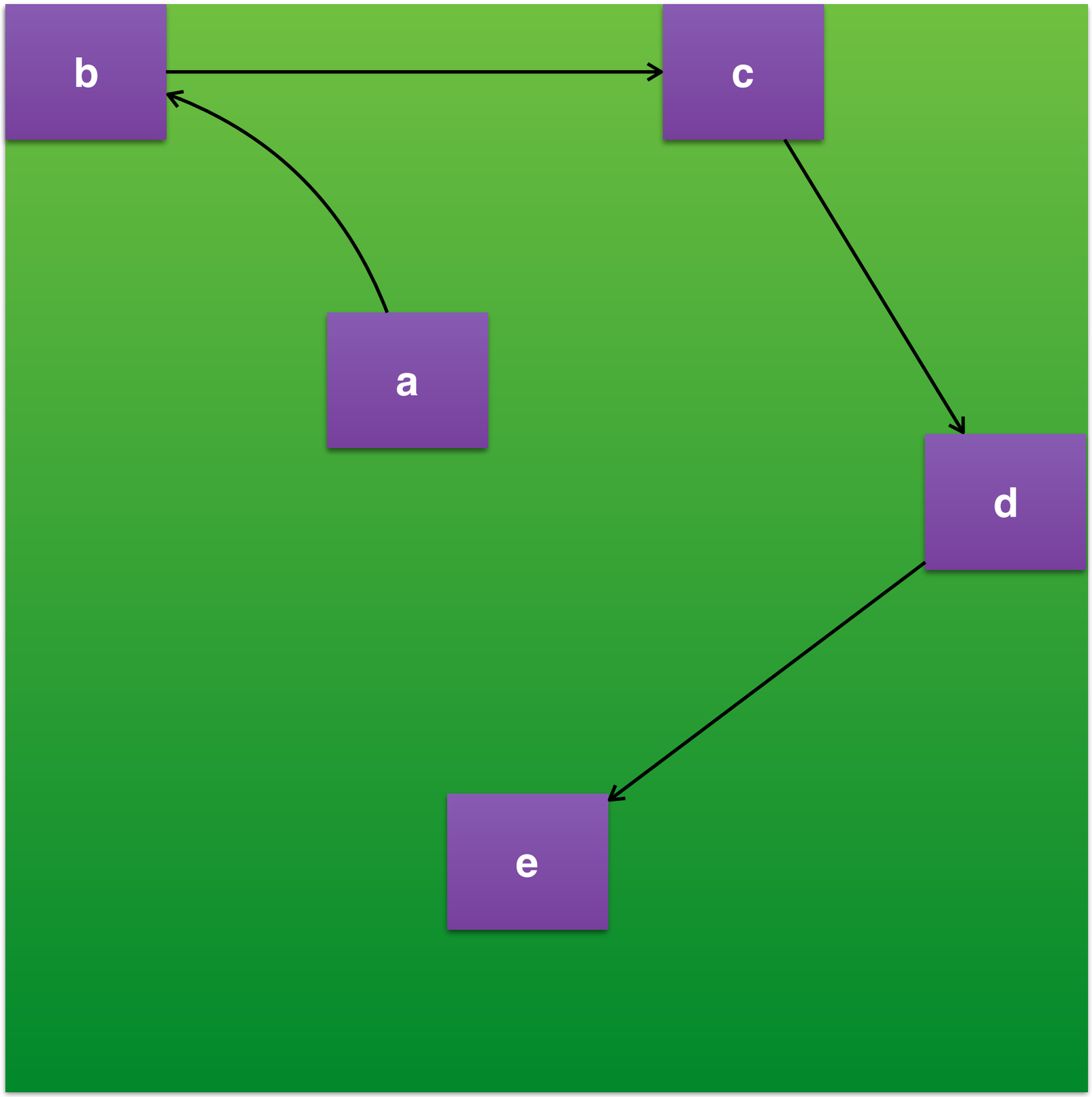
Pekare och länkade strukturer [På heapen]




```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 };
```

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

Att skapa noder i ett träd

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

```
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);
}
```

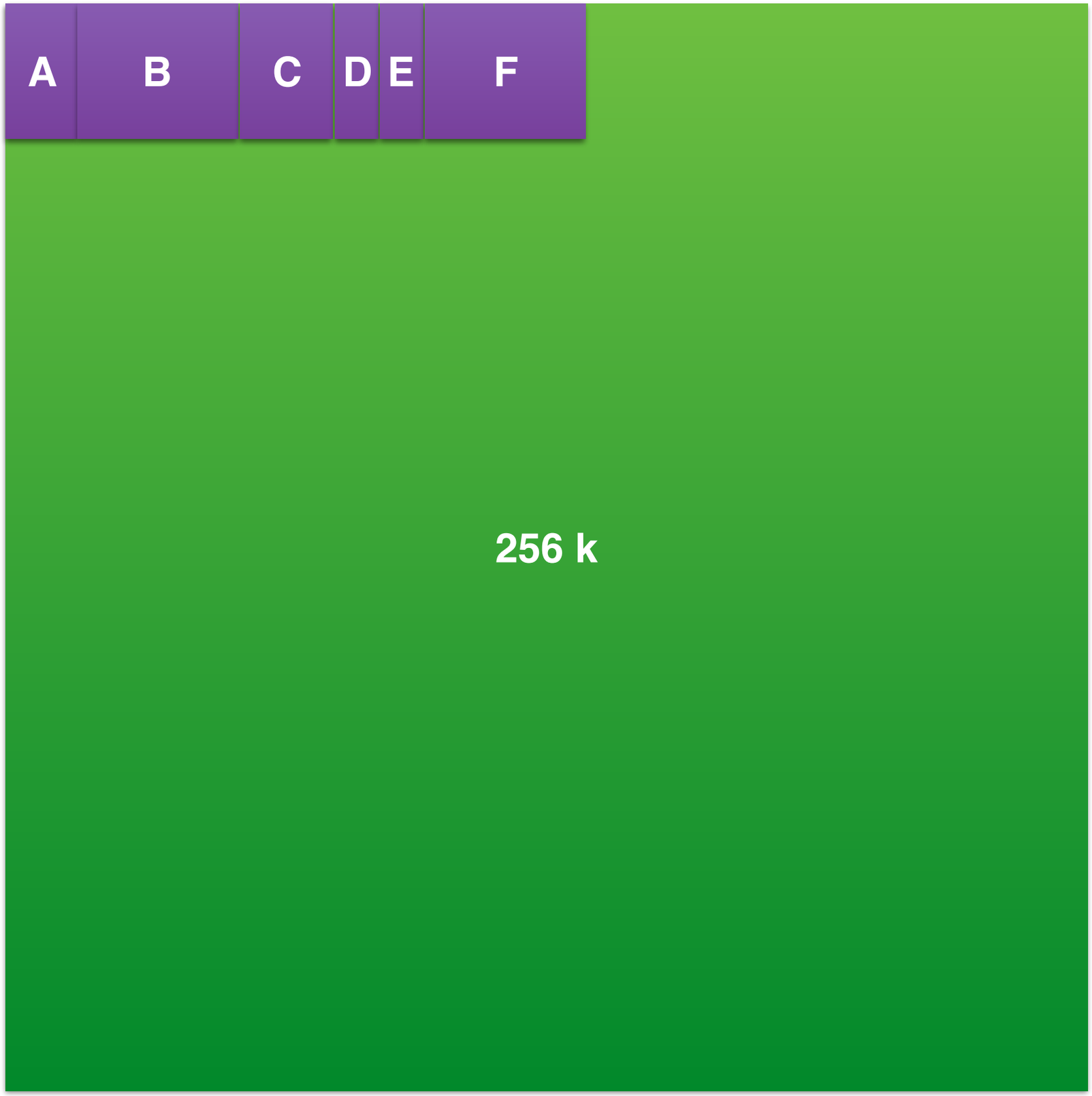
```
data_t *node_deallocate3(node_t *n)
{
  if (n->left)  free(n->left);
  if (n->right) free(n->right);
  free(n);
  return n->data; // BOOOOM!!!!
}
```

```
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;
}
```

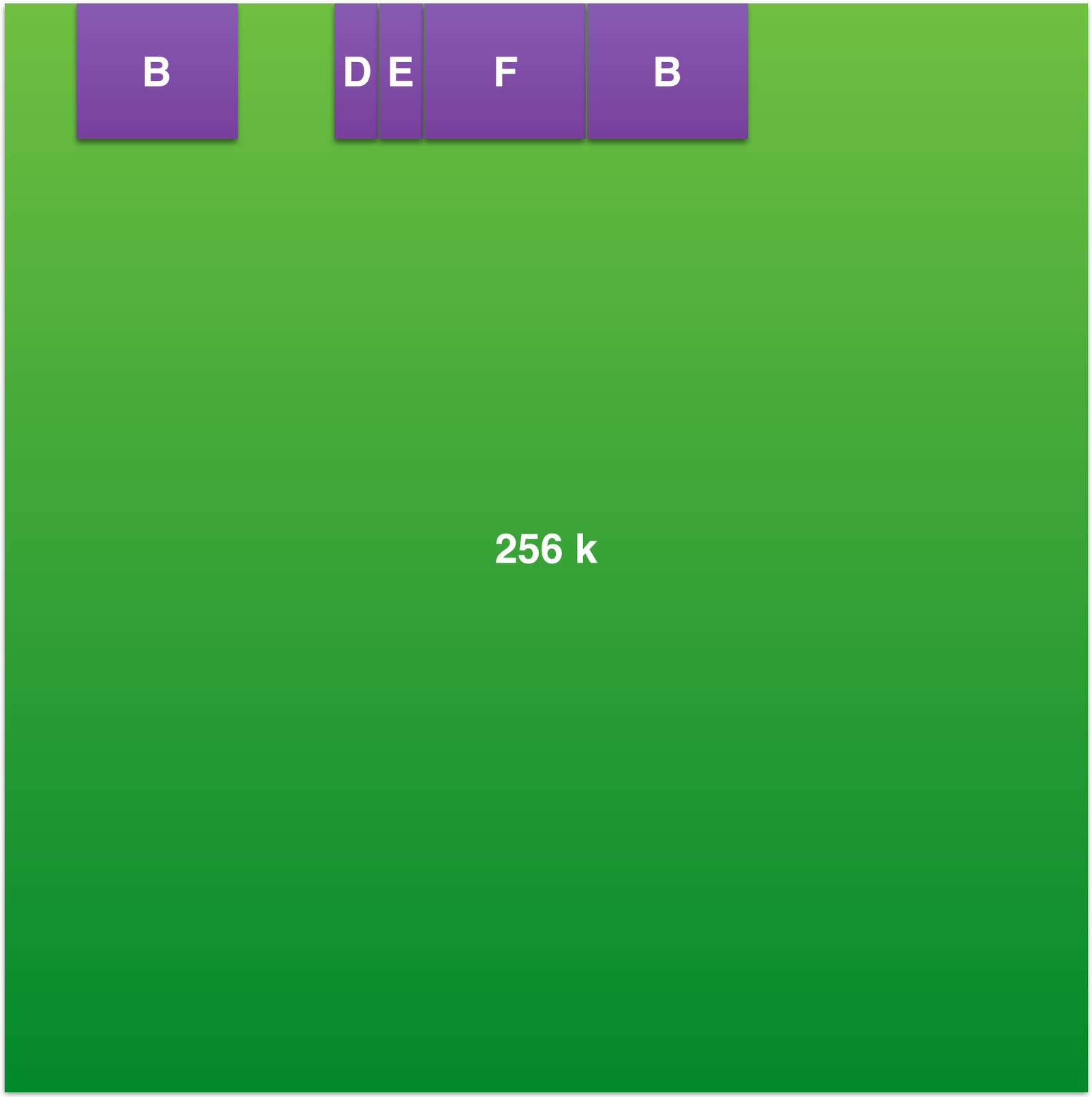
Minneshantering

- 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);
```



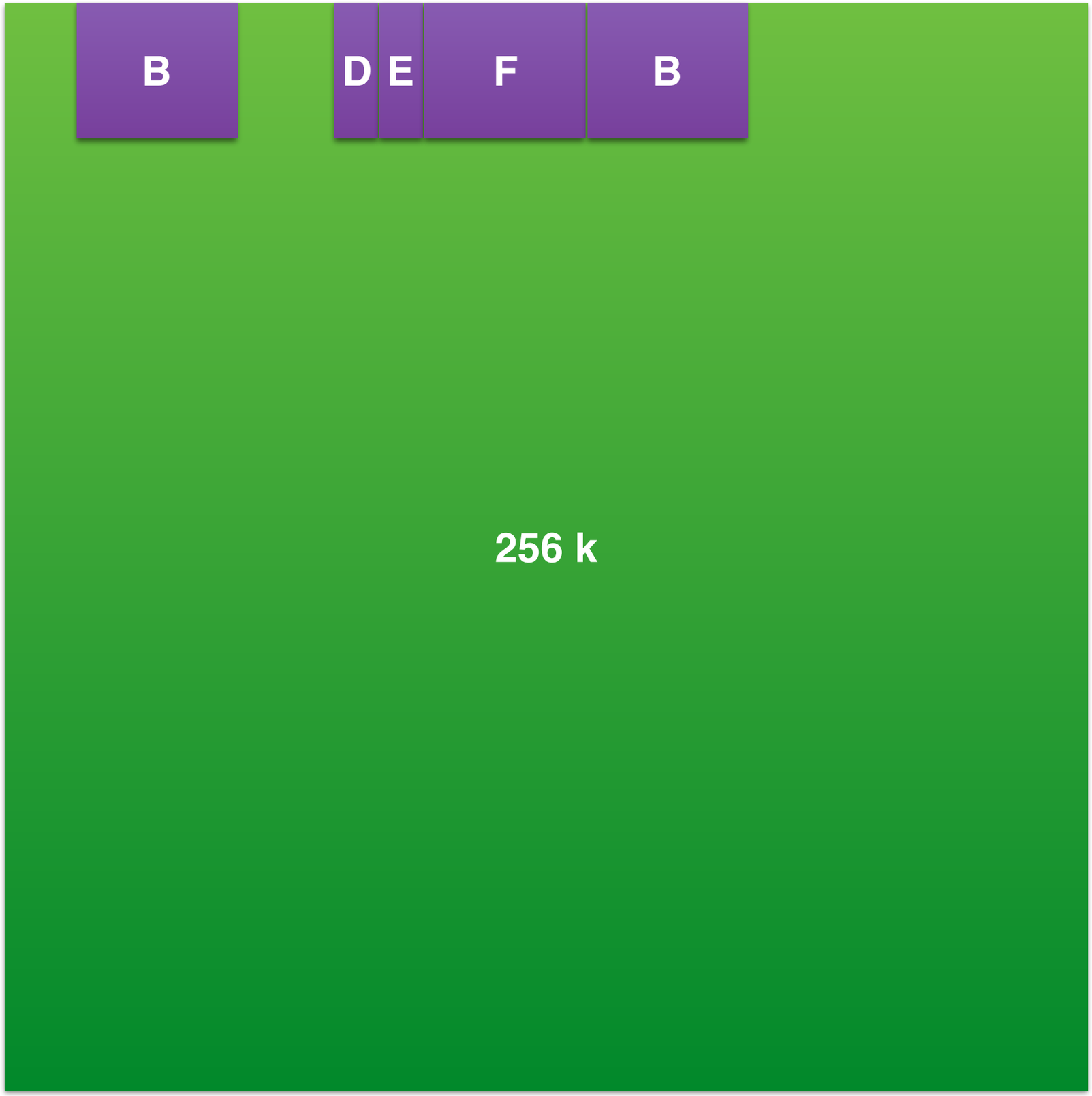
```
a = malloc(A);

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)

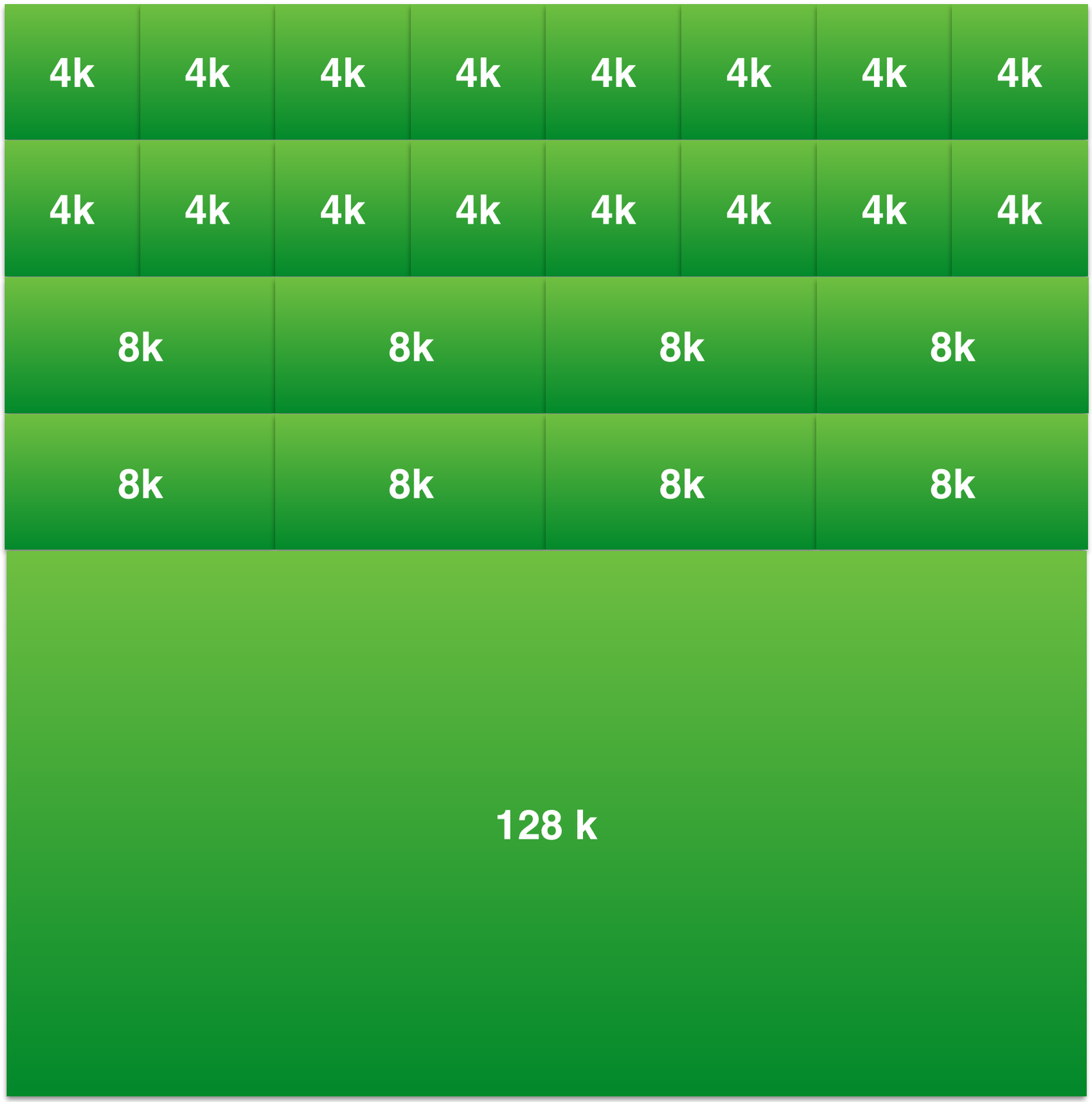
Kan leda till att minnet effektivt tar slut fast det finns gott om ledigt minne



Bucket Allocation

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

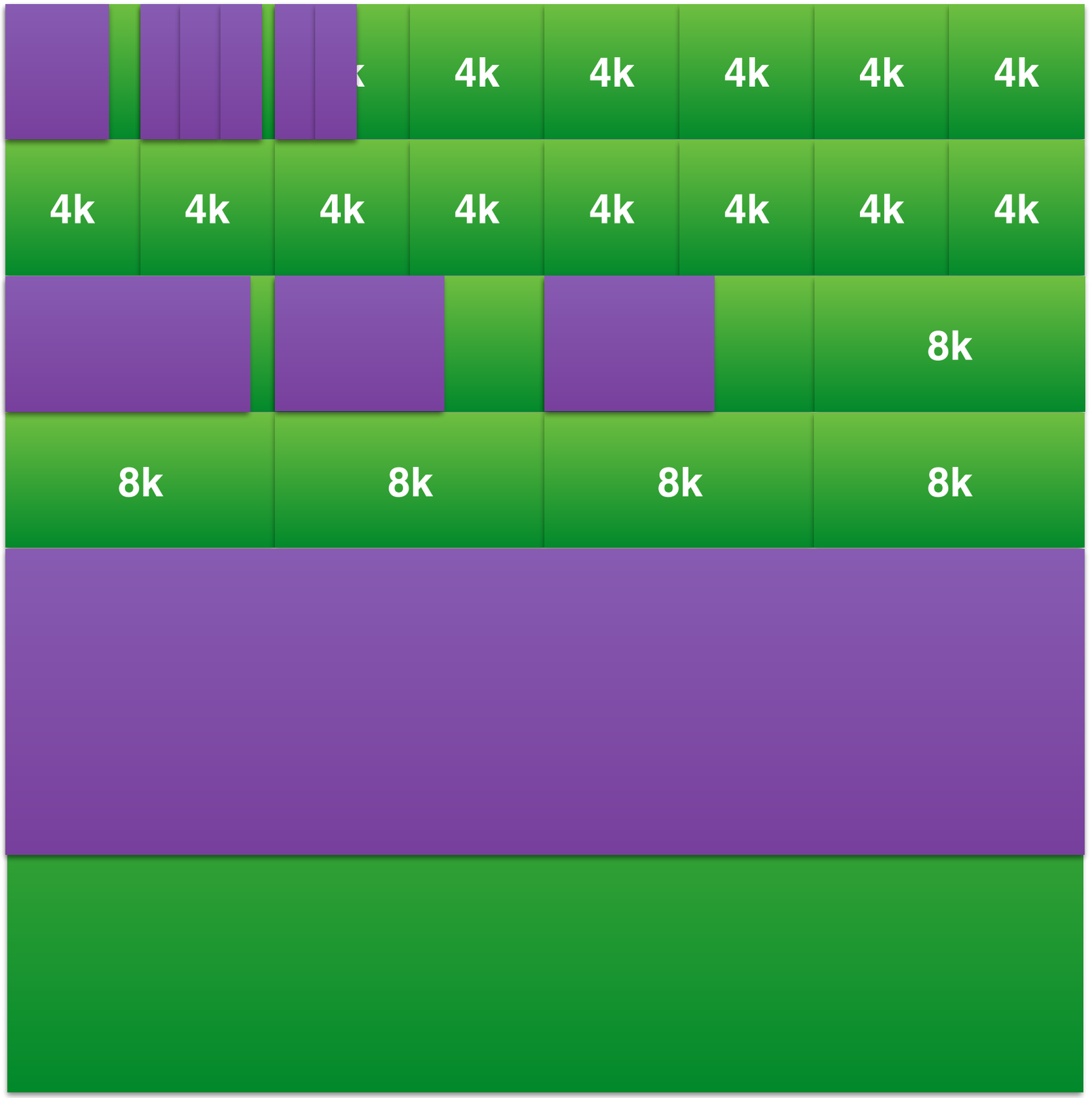
Vad får detta för effekt map fragmentering?



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 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

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 `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

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

```c
#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) prinf("%d ", sum[i]);

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

Hitta felet!

```c
#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) prinf("%d ", sum[i]);

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

1. Samma kod som ”sist”

2. even **och** odd är allokerade på stacken!

Hitta felet!

```c
#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)
    {
      prinf("%d ", sum[i]);
    }

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

```c
#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)
    {
      prinf("%d ", sum[i]);
    }

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

add
anropas
”med
samma
array”

Föreläsning 4

Tobias Wrigstad

Resten av kursen…

Så här långt borde du ha…

1. Gått på 3 föreläsningar
2. Gjort klart 5 av 6 labbar
3. Ha tittat runt på kursens webbsida
4. Ha skaffat ett GitHub-konto och registrerat det hos oss
5. Ha loggat in på Piazza minst en gång
6. …

Från och med nästa vecka: Labbar på IOOPM avser tid att redovisa och få hjälp med programmering.

Den mesta programmeringen sker utanför schemalagd tid!

Din uppgift

Att lära dig så mycket som möjligt om imperativ- och objektorienterad programmering, verktyg, program-meringsmetodik och process

Vår uppgift

Att hjälpa dig i din inlärningsprocess.

Att se till att det leder till att du blir godkänd på kursen.

IOOPM 2018

Tobias Wrigstad
Kursansvarig

Gustaf Borgström
Huvudassistent

Imperativ och ObjektOrienterad ProgrammeringsMetodik

Imperativ och ObjektOrienterad ProgrammeringsMetodik

Del 1 & 2

Imperativ och ObjektOrienterad ProgrammeringsMetodik

Del 2

Imperativ och ObjektOrienterad ProgrammeringsMetodik

Del 1, 2 & 3

Efter godkänd kurs ska studenten kunna [1/2]

- förklara hur program **exekverar** och beskriva hur program **lagrar och hanterar information**.
- förklara och exemplifiera **kvalitativa** och **kvantitativa aspekter** av ett programs **design** och **implementation**.
- förklara skillnaden mellan **manuell och automatisk minneshantering** samt grundläggande relaterade begrepp (som stack, heap, statisk minnesarea) och använda **verktyg** för felsökning av minnesfel.
- förklara hur **större programuppgifter** kan lösas & resonera om olika lösningsalternativ.
- redogöra för hur **enkla parallelliserbara problem kan lösas** effektivt med relevanta hjälpmedel.
- **designa, koda, granska, testa, felsöka och dokumentera** egna program, med hjälp av lämpliga **verktyg**.
- **läsa, förstå och modifiera** icke-trivial **kod** som studenten själv inte har skrivit samt **integrera nyskriven kod med existerande**.

Efter godkänd kurs ska studenten kunna [2/2]

- beskriva – **oberoende av programkoden** – den uppgift ett program skall lösa och de förutsättningar som krävs för att det skall kunna arbeta. Motivera varför programmet under dessa förutsättningar är korrekt, **samt specificera och konstruera testfall och köra tester för att verifiera detta**.
- skriva lämplig **dokumentation för programmering** och **testhantering**.
- tillämpa specifika **utsnitt av kända utvecklingsprocesser** och -metodiker (ex. **agil utveckling**, **parprogrammering** och testdriven utveckling).
- beskriva olika former av **testning** och deras vikt i olika skeden av **utvecklings-processen**.
- bidra till ett **konstruktivt samarbete i programmeringsprojekt**.
- **presentera** och **diskutera** kursens innehåll **muntligt** och **skriftligt** med för utbildningsnivån lämplig färdighet.

Mål

Unlockable Achievements (aka kursmål/kunskapsmål)

Name	Short desc	Grade level	Assessment
A1	Konsekvent tillämpa procedurell abstraktion för att öka läsbarheten och undvika upprepningar	3	L
A2	Tillämpa objektorienterad abstraktion för att dölja implementationsdetaljer bakom väldefinierade gränssnitt	3	L
A3	Demonstrera förståelse för designprincipen informationsgömning i ett C-program med hjälp av .c och .h-filer	4	L, G
B4	Använda arv, metodspecialisering och superanrop i ett program som drar nytta av subtypspolymorfism	3	L

ID

Ingress

Nivå

3
4
5

Hur målet kan redovisas

L — Labb
W — Essä
T — Tobias
P — Presentation
R — Rapport

wrigstad.com/ioopm

Mål

Unlockable Achievemen kursmål/kunskapsmål)

Name	Short desc
A1	Konsekvent tillämpa procedurell abstraktion för a och undvika upprepningar
A2	Tillämpa objektorienterad abstraktion för att dölja implementationsdetaljer bakom väldefinierade grä
A3	Demonstrera förståelse för designprincipen inform ett C-program med hjälp av .c och .h-filer
B4	Använda arv, metodspecialisering och superanrop som drar nytta av subtypspolymorfism

(A. Abstraktion)

A1

Konsekvent tillämpa procedurell abstraktion för att öka läsbarheten och undvika upprepningar

Level	Assessment
3	L

Abstraktion är en av de viktigaste programmeringsprinciperna. Vi vet att djupt, djupt nere under huven är allt bara ettor och nollor (redan detta är en abstraktion!), men ovanpå dessa har vi byggt lager på lager av abstraktioner som låter oss tala om program t.ex. i termer av struktar och procedurer.

Proceduren ritaEnCirkel(int radie, koordinat center) utför beräkningar och tänder individuella pixlar på en skärm, men i och med att dessa rutiner kapslats in i en procedur med ett vettigt namn, där indata är uttryckt i termer av koordinater och radie har vi abstraherat bort dessa detaljer, och det blir möjligt att rita cirklar tills korna kommer hem utan att förstå hur själva implementationen ser ut.

Väl utförd abstraktion döljer detaljer och låter oss fokusera på färre koncept i taget.

Du bör ha en klar uppfattning om bland annat:

- Varför det är vettigt att identifiera liknande mönster i koden och extrahera dem och kapsla in dem i en enda procedur som kan anropas istället för upprepningarna?
- Abstraktioner kan "läcka". Vad betyder det och vad får det för konsekvenser?
- Vad är skillnaderna mellan "control abstraction" och "data abstraction"? (Du kan läsa om dessa koncept på t.ex. Wikipedia).

Notera: målens namn/nummer kan ha ändrats

Redovisning

- **Din uppgift:** att övertyga examinatorn om att du uppfyller målet

  Ge **exempel** från den kod (etc.) du har skrivit

  Inga **core dumps**

  Ha en **story** för hur allting hänger ihop

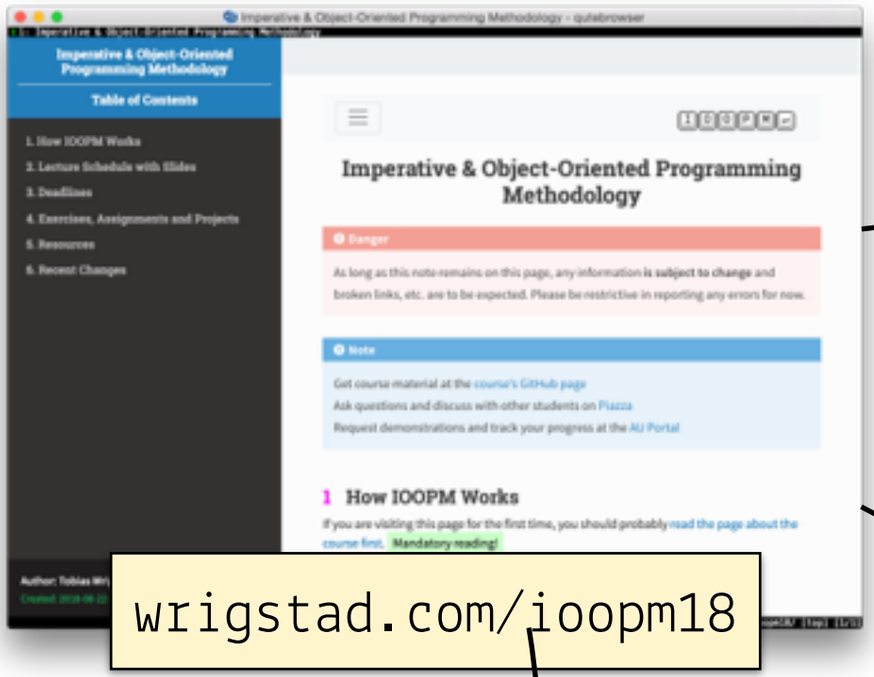
- Försök att alltid redovisa **> 1 mål åt gången**

  Mindre arbete, mindre väntetid

  Synergi!

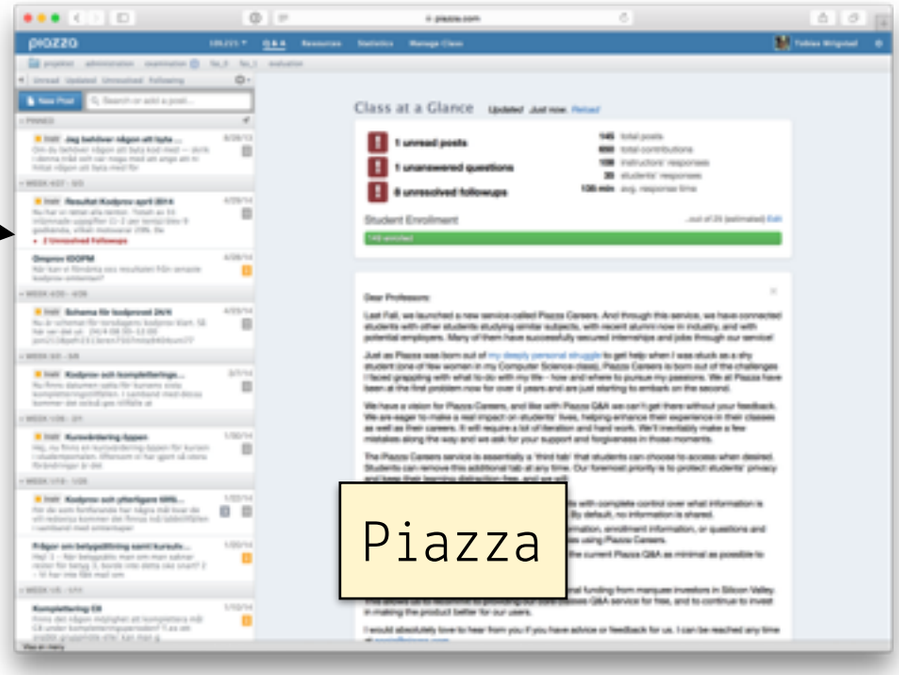
- **Demo av redovisningsystemet sker på föreläsning innan första redovisningstillfället**

Var finns information om kursen?



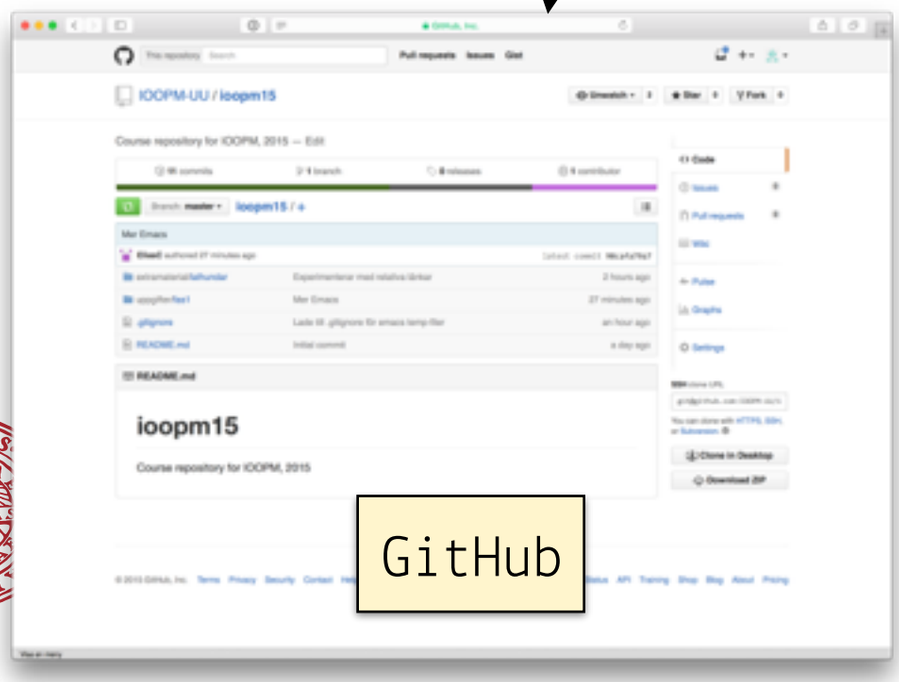
wrigstad.com/ioopm18

Hur interagerar jag med er?



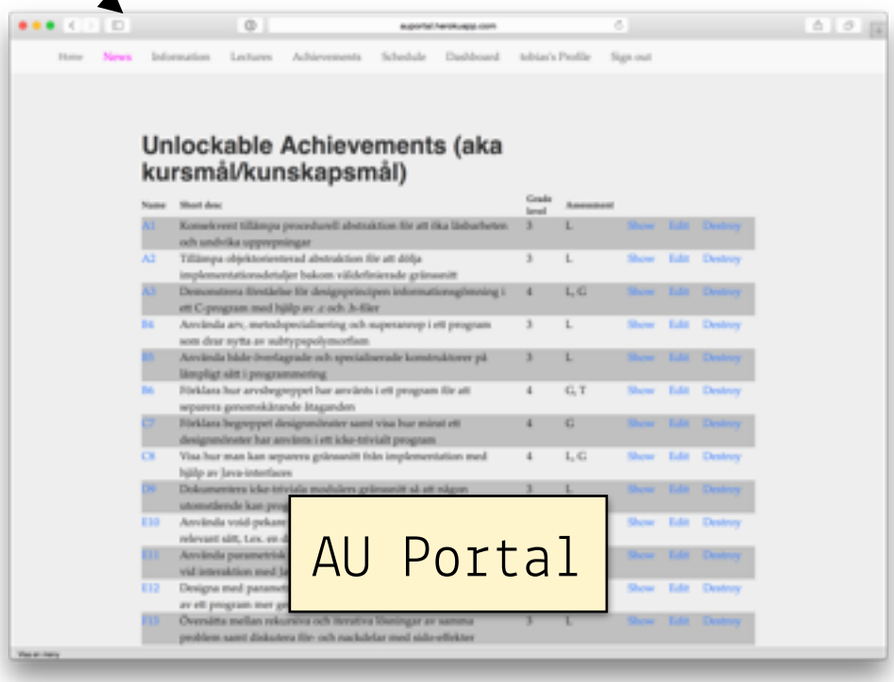
Piazza

Hur lämnar jag in en uppgift?



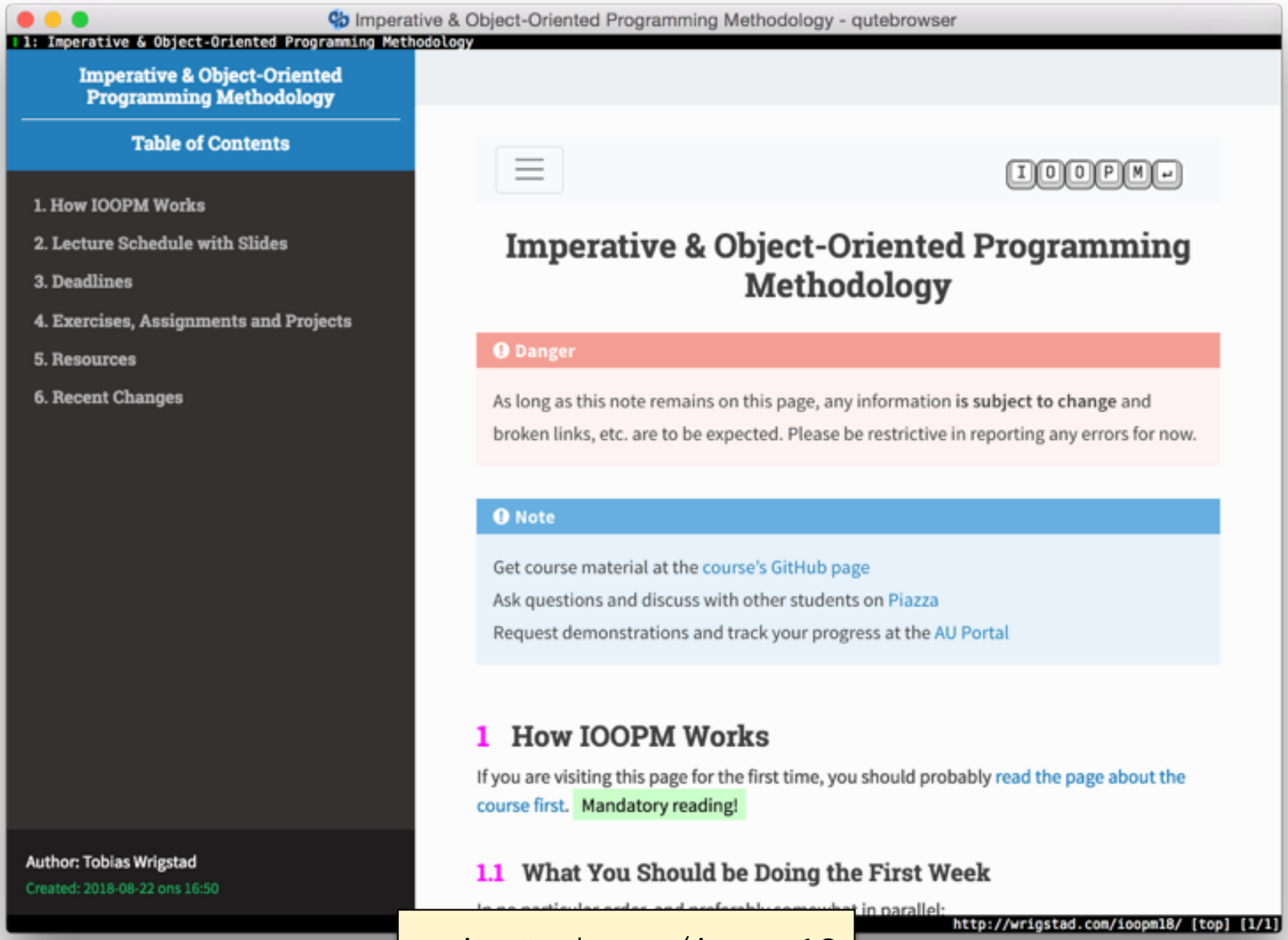
GitHub

Hur redovisar jag?



AU Portal

Du börjar här:



wrigstad.com/ioopm18

Översikt

Ritas på tavlan

Fas [kursen har 3]

Gör en plan

UM!

2 veckors sprint

Revidera!

Gör en plan

UM!

2 veckors sprint

Revidera!

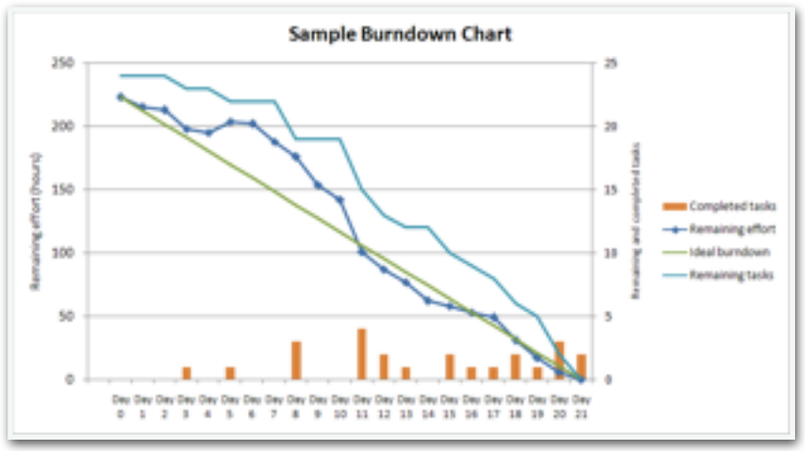
Gör en plan

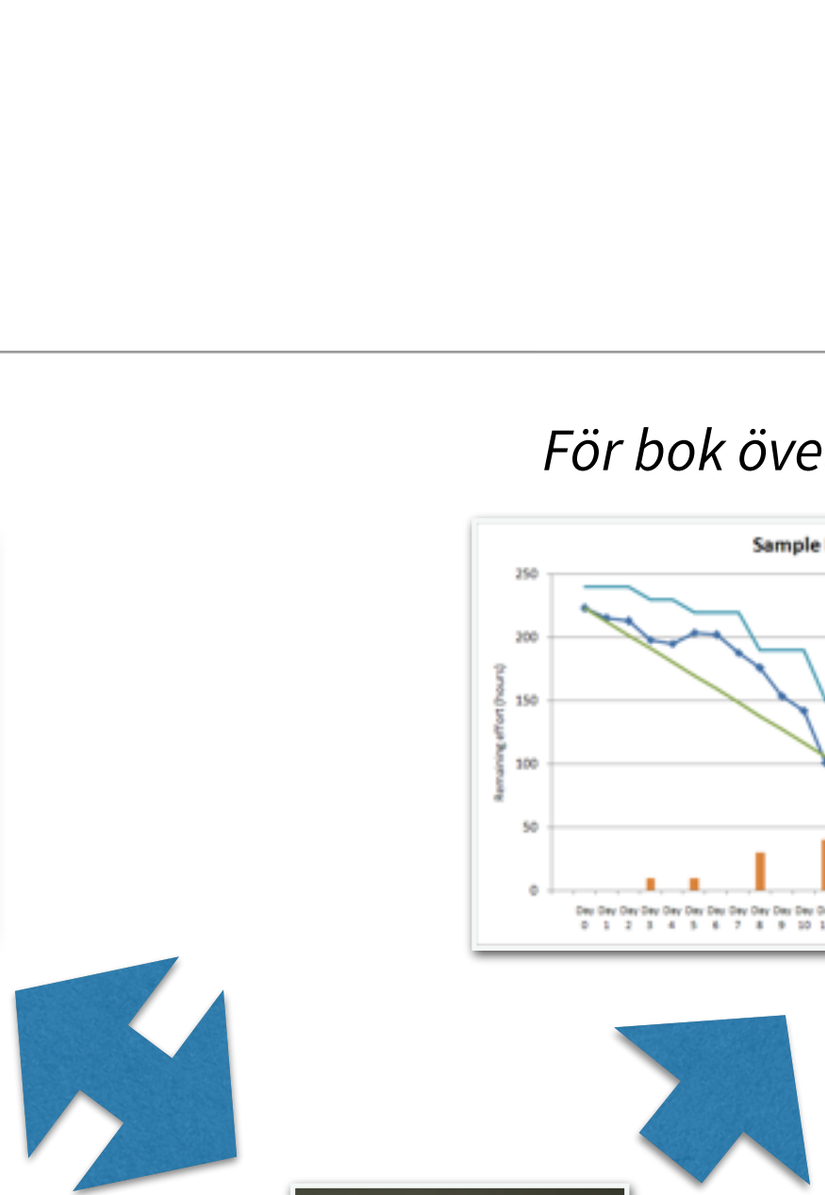
Sprint [varje fas har 2]

Uppgift

Implementera

För bok över dina framsteg





Välj mål

Redovisa

Uppföljningsmöte

- I slutet av varje sprint

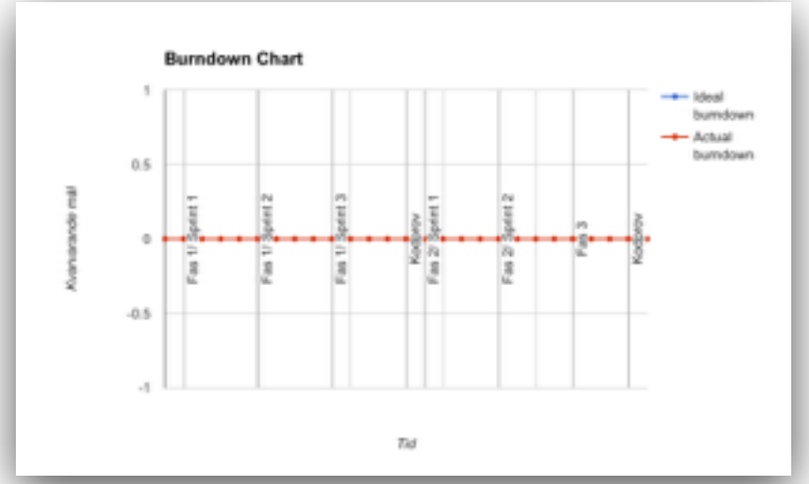
  Diskussion kring och utifrån burndown chart

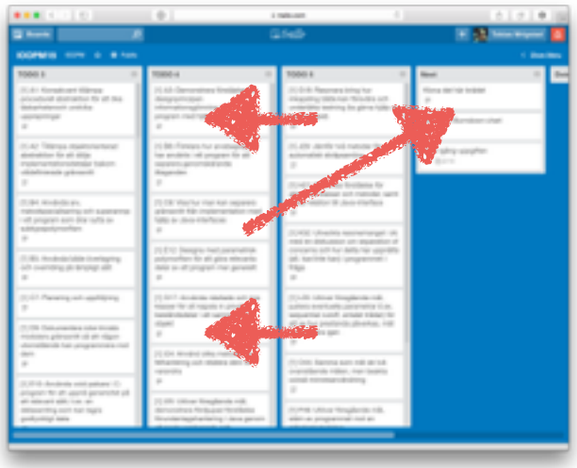
  Tillsammans med andra studenter

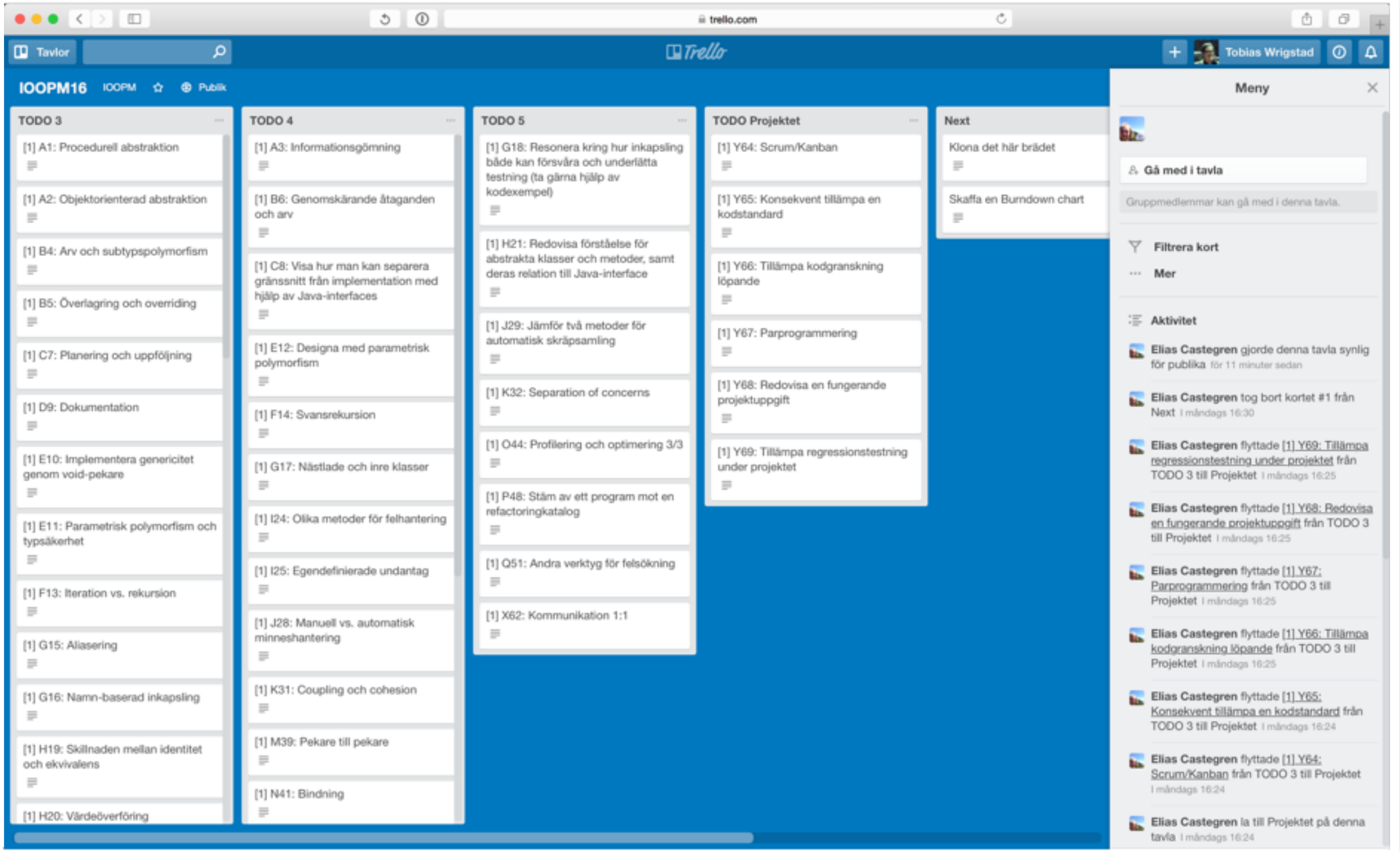
  I början av kursen är det fokus på planering

  Vi går mot fokus på uppföljning

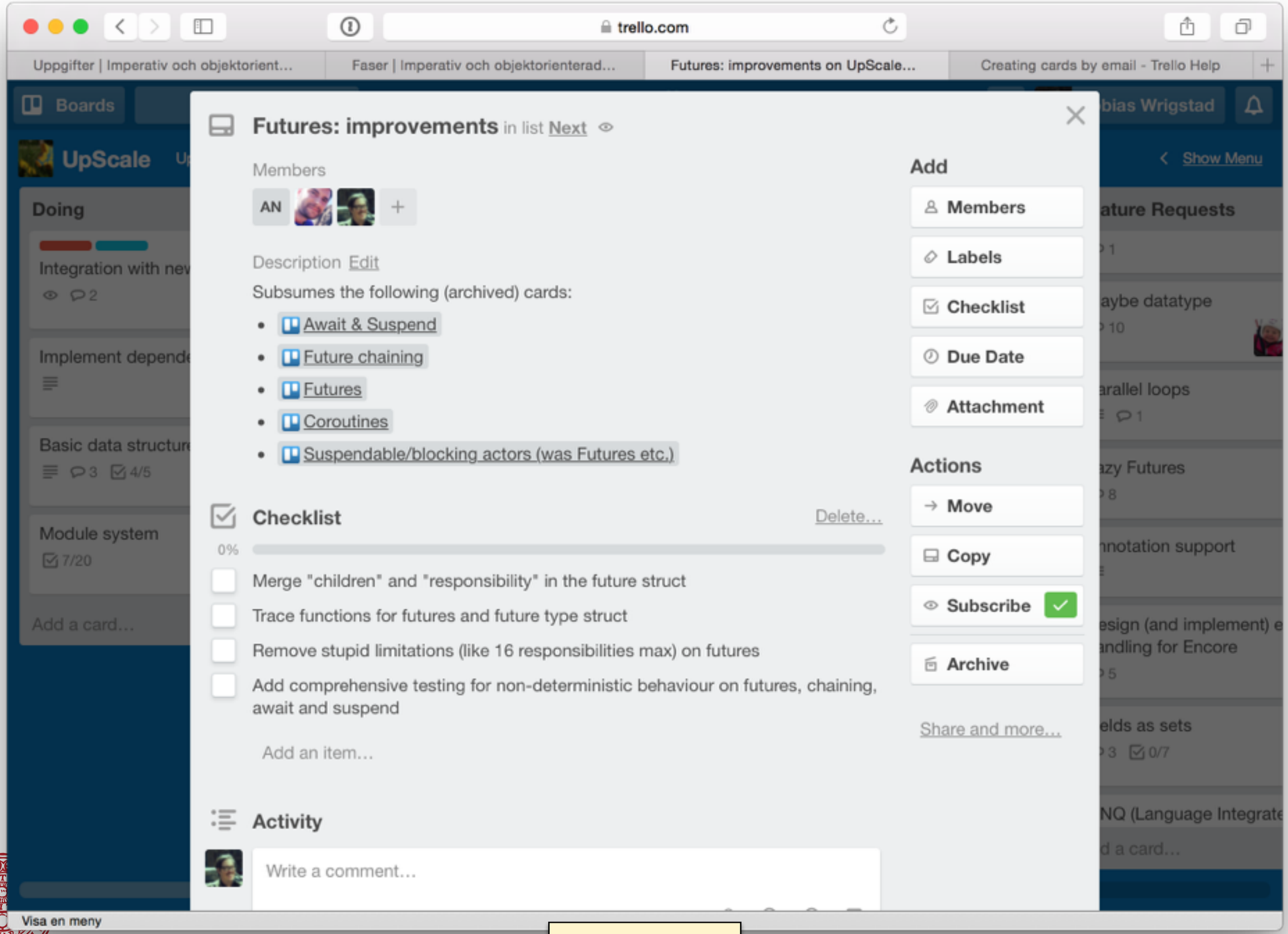
- **Nästa möte nu på fredag**






trello.com
Tavlor
Trello
Tobias Wrigstad
IOOPM16
IOOPM
Publik
TODO 3
[1] A1: Procedurell abstraktion
[1] A2: Objektorienterad abstraktion
[1] B4: Arv och subtypspolymorfism
[1] B5: Överlagring och overriding
[1] C7: Planering och uppföljning
[1] D9: Dokumentation
[1] E10: Implementera genericitet genom void-pekare
[1] E11: Parametrisk polymorfism och typsäkerhet
[1] F13: Iteration vs. rekursion
[1] G15: Aliasering
[1] G16: Namn-baserad inkapsling
[1] H19: Skillnaden mellan identitet och ekvivalens
[1] H20: Värdeöverföring
TODO 4
[1] A3: Informationsgömning
[1] B6: Genomskärande åtaganden och arv
[1] C8: Visa hur man kan separera gränssnitt från implementation med hjälp av Java-interfaces
[1] E12: Designa med parametrisk polymorfism
[1] F14: Svansrekursion
[1] G17: Nästlade och inre klasser
[1] I24: Olika metoder för felhantering
[1] I25: Egendefinierade undantag
[1] J28: Manuell vs. automatisk minneshantering
[1] K31: Coupling och cohesion
[1] M39: Pekare till pekare
[1] N41: Bindning
TODO 5
[1] G18: Resonera kring hur inkapsling både kan försvåra och underlätta testning (ta gärna hjälp av kodexempel)
[1] H21: Redovisa förståelse för abstrakta klasser och metoder, samt deras relation till Java-interface
[1] J29: Jämför två metoder för automatisk skräpsamling
[1] K32: Separation of concerns
[1] O44: Profilering och optimering 3/3
[1] P48: Stäm av ett program mot en refactoringkatalog
[1] Q51: Andra verktyg för felsökning
[1] X62: Kommunikation 1:1
TODO Projektet
[1] Y64: Scrum/Kanban
[1] Y65: Konsekvent tillämpa en kodstandard
[1] Y66: Tillämpa kodgranskning löpande
[1] Y67: Parprogrammering
[1] Y68: Redovisa en fungerande projektuppgift
[1] Y69: Tillämpa regressionstestning under projektet
Next
Klona det här brädet
Skaffa en Burndown chart
Meny
Gå med i tavla
Gruppmedlemmar kan gå med i denna tavla.
Filtrera kort
Mer
Aktivitet
Elias Castegren gjorde denna tavla synlig för publika för 11 minuter sedan
Elias Castegren tog bort kortet #1 från Next I måndags 16:30
Elias Castegren flyttade [1] Y69: Tillämpa regressionstestning under projektet från TODO 3 till Projektet I måndags 16:25
Elias Castegren flyttade [1] Y68: Redovisa en fungerande projektuppgift från TODO 3 till Projektet I måndags 16:25
Elias Castegren flyttade [1] Y67: Parprogrammering från TODO 3 till Projektet I måndags 16:25
Elias Castegren flyttade [1] Y66: Tillämpa kodgranskning löpande från TODO 3 till Projektet I måndags 16:25
Elias Castegren flyttade [1] Y65: Konsekvent tillämpa en kodstandard från TODO 3 till Projektet I måndags 16:24
Elias Castegren flyttade [1] Y64: Scrum/Kanban från TODO 3 till Projektet I måndags 16:24
Elias Castegren la till Projektet på denna tavla I måndags 16:24

trello.com


trello.com
Uppgifter | Imperativ och objektorient...
Faser | Imperativ och objektorienterad...
Futures: improvements on UpScale...
Creating cards by email - Trello Help
Boards
UpScale
Show Menu
Doing
Integration with new
Implement depende
Basic data structure
Module system
7/20
Add a card...
Futures: improvements in list Next
Members
AN
Description Edit
Subsumes the following (archived) cards:
Await & Suspend
Future chaining
Futures
Coroutines
Suspendable/blocking actors (was Futures etc.)
Checklist
Delete...
0%
Merge "children" and "responsibility" in the future struct
Trace functions for futures and future type struct
Remove stupid limitations (like 16 responsibilities max) on futures
Add comprehensive testing for non-deterministic behaviour on futures, chaining, await and suspend
Add an item...
Activity
Write a comment...
Add
Members
Labels
Checklist
Due Date
Attachment
Actions
Move
Copy
Subscribe
Archive
Share and more...
Visa en meny

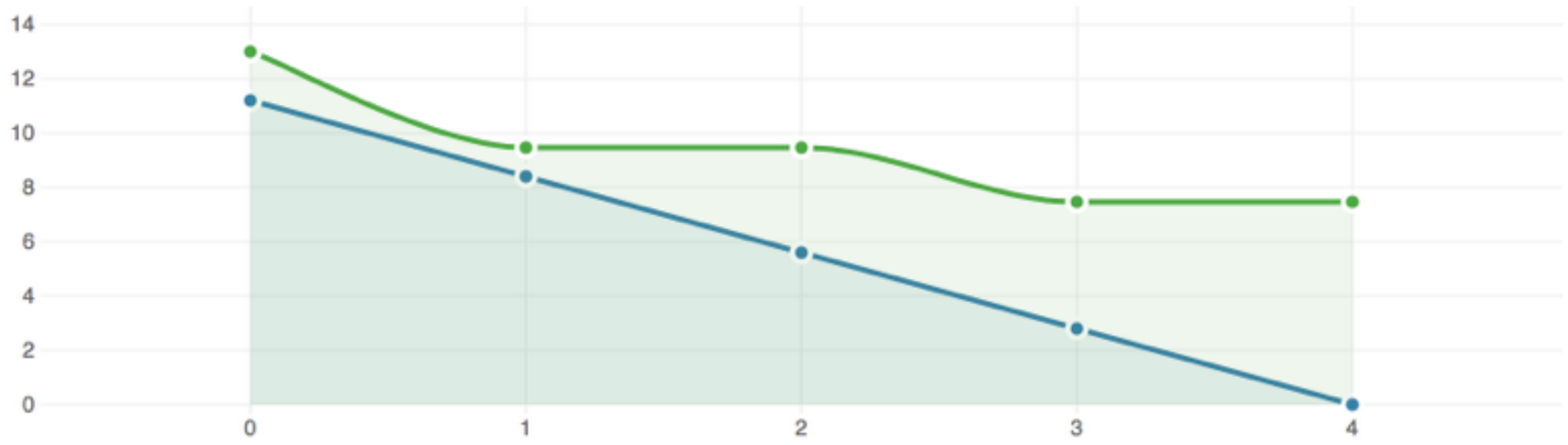
trello.com

Uppgifter | Imperativ och objektorient... | Faser | Imperativ och objektorienterad... | Work burndown chart | Creating cards by email - Trello Help

Work burndown chart | Home | View sprints | Update active sprints | Manage sprints | About

What's on this page? You are currently viewing statistics of sprint **4.**

Burndown chart



- green: done estimate
- blue: ideal estimate

Done estimate vs. effort (daily)

Visa en meny

https://github.com/devtyr/trello-burndown

docs.google.com

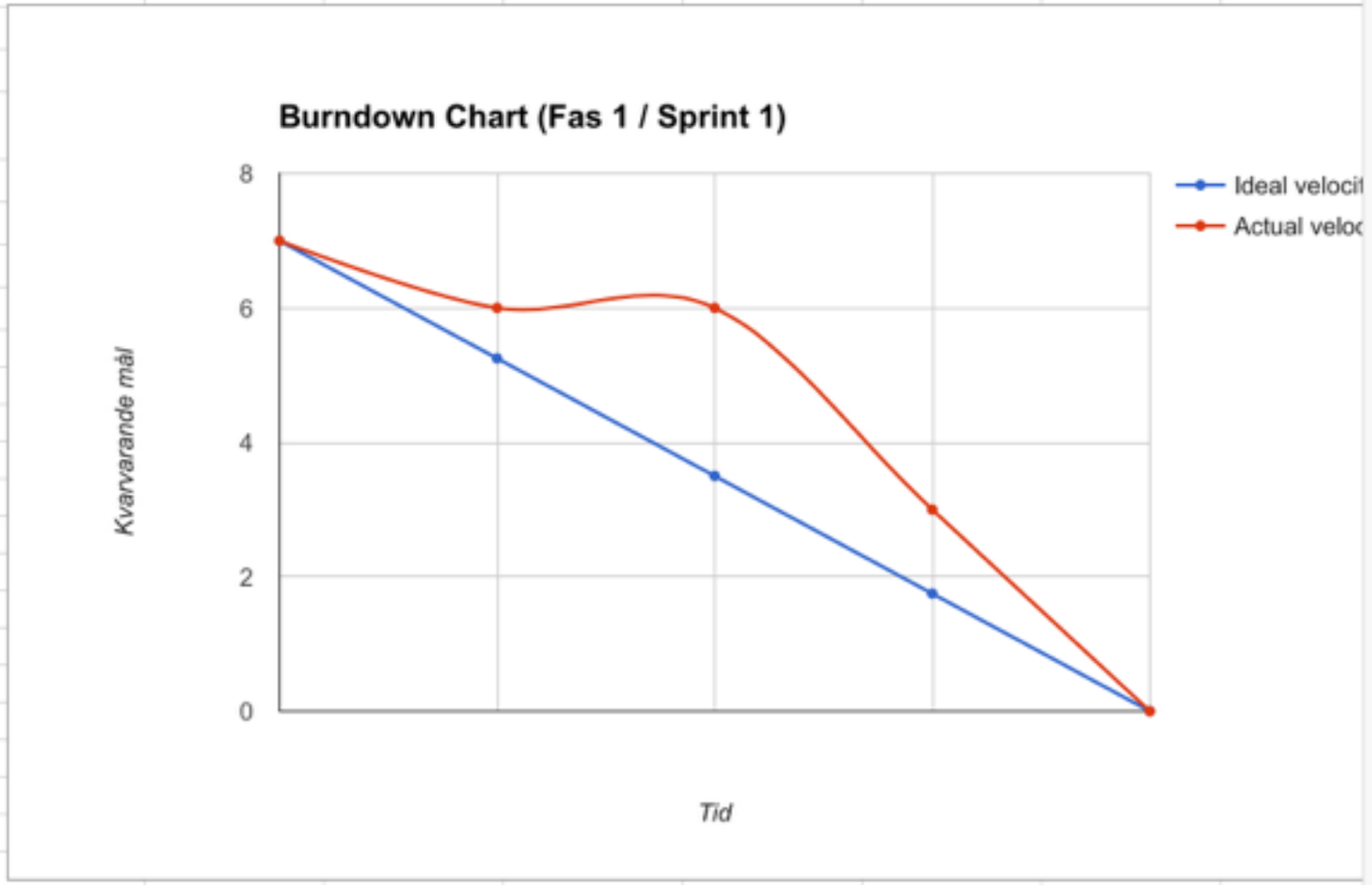
Copy of [IOOPM] Burndown chart

File Edit View Insert Format Data Tools Add-ons Help All changes saved in Drive

tobias.wrigstad@gmail.com

Comments Share

	Hur många mål siktar du på att ta denna sprint?	7
	Hur många mål har du tagit?	
	Lab 2	1
	Lab 3	0
	Lab 4	3
	Lab 5	3



Burndown totalt | Fas1/Sprint1 | Fas1/Sprint2 | Fas1/Sprint3 | Fas2/Sprint1 | Fas2/Sprint2 | Fas3 | Burndown

Visa en meny

Se ”Länkar” på kurswebben

Högskolepoäng

	HP	Deadline	
Fas 1	5	V44*	
Fas 2	5	V49	
Fas 3	5	V2	
Kodprov	2,5	*	Oktober
	2,5	*	December

**It's complicated (se kurswebben för detaljer)*

Övning i skriftlig färdighet

- På nivå 4 och 5 måste du redovisa ett mål som en essä

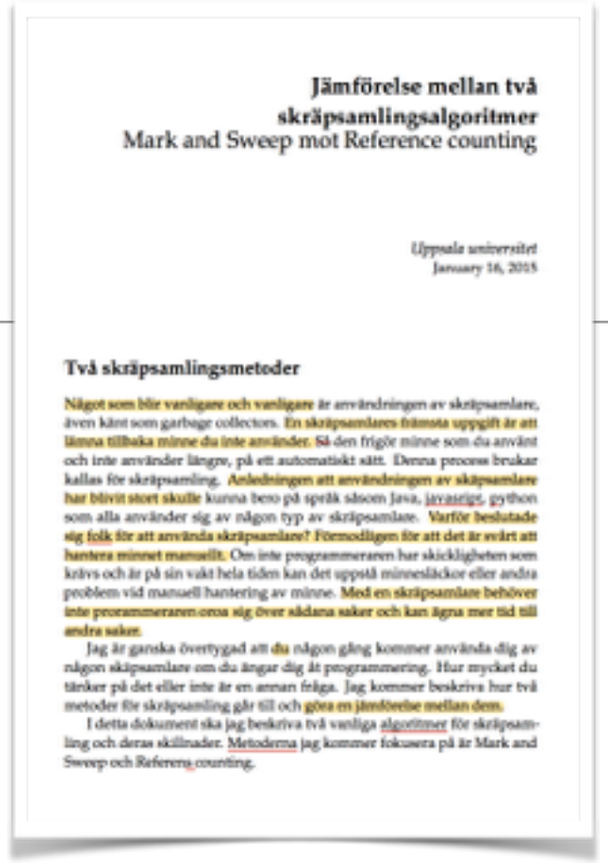
  (För att nå nivå 3 räcker det med projektrapporten)

- Instruktioner finns på kurswebben

  Omfattning: 7500 tecken

  Deadline: **se kurswebben**

- Lämnas in via GitHub



Jämförelse mellan två skräpsamlingsalgoritmer
Mark and Sweep mot Reference counting

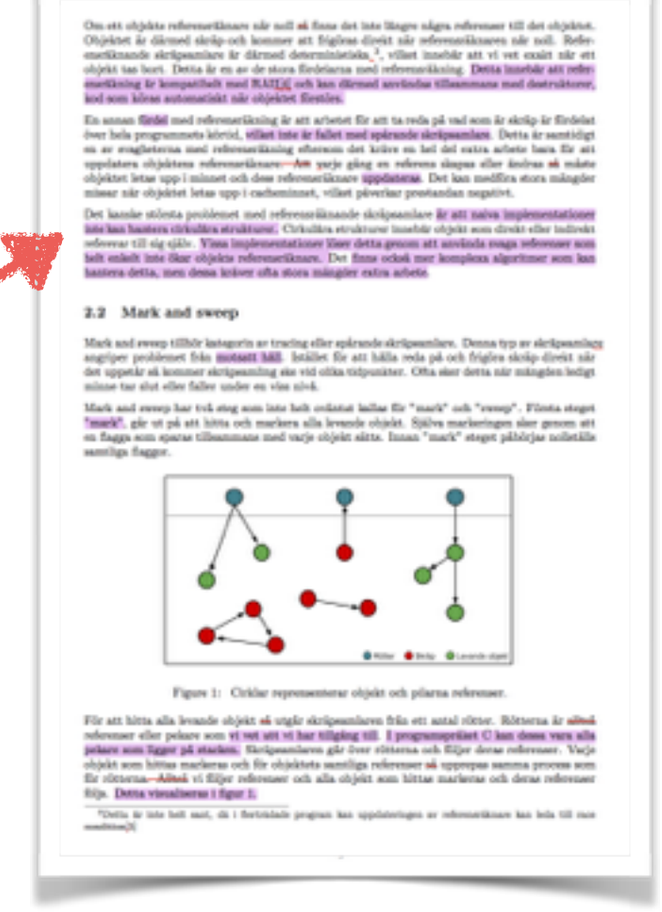
Uppsala universitet
January 16, 2015

Två skräpsamlingsmetoder

Något som blir vanligare och vanligare är användningen av skräpsamlare, även känt som garbage collectors. En skräpsamlares främsta uppgift är att lämna tillbaka minne du inte använder. Så den frigör minne som du använt och inte använder längre, på ett automatiskt sätt. Denna process brukar kallas för skräpsamling. Anledningen att användningen av skräpsamlare har blivit stort skulle kunna bero på språk såsom Java, javascript, python som alla använder sig av någon typ av skräpsamlare. Varför beslutade sig folk för att använda skräpsamlare? Förmodligen för att det är svårt att hantera minnet manuellt. Om inte programmeraren har skickligheten som krävs och är på sin vakt hela tiden kan det uppstå minnesläckor eller andra problem vid manuell hantering av minne. Med en skräpsamlare behöver inte programmeraren oroa sig över sådana saker och kan ägna mer tid till andra saker.

Jag är ganska övertygad att du någon gång kommer använda dig av någon skräpsamlare om du ängar dig åt programmering. Hur mycket du tänker på det eller inte är en annan fråga. Jag kommer beskriva hur två metoder för skräpsamling går till och göra en jämförelse mellan dem.

I detta dokument ska jag beskriva två vanliga algoritmer för skräpsamling och deras skillnader. Metoderna jag kommer fokusera på är Mark and Sweep och Referens counting.



Om ett objekts referensräknare når noll så finns det inte längre några referenser till det objektet. Objektet är därmed skräp och kommer att frigöras direkt när referensräknaren når noll. Referensräknande skräpsamlare är därmed deterministiska,[1], vilket innebär att vi vet exakt när ett objekt tas bort. Detta är en av de stora fördelarna med referensräkning. Detta innebär att referensräkning är kompatibelt med RAII[2] och kan därmed användas tillsammans med destruktorer, kod som körs automatiskt när objektet försvinner.

En annan fördel med referensräkning är att arbetet för att ta reda på vad som är skräp är fördelat över hela programmets körtid, vilket inte är fallet med spårande skräpsamlare. Detta är samtidigt en av svagheterna med referensräkning eftersom det kräve en hel del extra arbete bara för att uppdatera objektens referensräknare—Att varje gång en referens skapas eller ändras så måste objektet letas upp i minnet och dess referensräknare uppdateras. Det kan medföra stora mängder minnes när objektet letas upp i cacheminnet, vilket påverkar prestandan negativt.

Det kanske största problemet med referensräknande skräpsamlare är att naiva implementationer inte kan hantera cirkulära strukturer. Cirkulära strukturer innebär objekt som direkt eller indirekt refererar till sig själv. Vissa implementationer löser detta genom att använda svaga referenser som helt enkelt inte ökar objekts referensräknare. Det finns också mer komplexa algoritmer som kan hantera detta, men dessa kräver ofta stora mängder extra arbete.

2.2 Mark and sweep

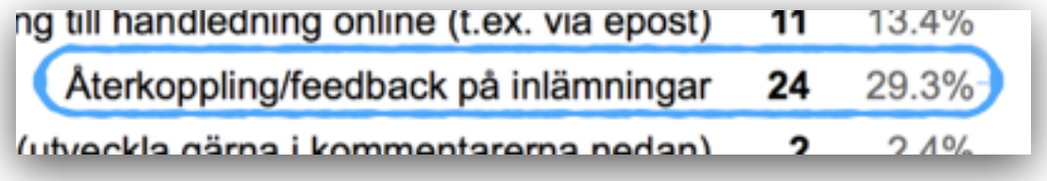
Mark and sweep tillhör kategorin av tracing eller spårande skräpsamlare. Denna typ av skräpsamlare angriper problemet från motsatt håll. Istället för att hålla reda på och frigöra skräp direkt när det uppstår så kommer skräpsamling ske vid olika tidpunkter. Ofta sker detta när mängden ledigt minne tar slut eller faller under en viss nivå.

Mark and sweep har två steg som inte helt oväntat heter för "mark" och "sweep". Första steget "mark", går ut på att hitta och markera alla levande objekt. Själva markeringen sker genom att en flagga som sparas tillsammans med varje objekt sätts. Innan "mark" steget påbörjas nollställs samtliga flaggor.

Figure 1: Cirklar representerar objekt och pilarna referenser.

För att hitta alla levande objekt så utgår skräpsamlaren från ett antal rötter. Rötterna är alltså referenser eller pekare som vi vet att vi har tillgång till. I programspråket C kan dessa vara alla pekare som ligger på stacken. Skräpsamlaren går över rötterna och följer deras referenser. Varje objekt som hittas markeras och för objektets samtliga referenser så upprepas samma process som för rötterna. Alltså vi följer referenser och alla objekt som hittas markeras och deras referenser följs. Detta visualiseras i figur 1.

[1]Detta är inte helt sant, då i flertalende program kan uppdateringen av referensräknare kan leda till rare



ng till handledning online (t.ex. via epost)	11	13.4%
Återkoppling/feedback på inlämningar	24	29.3%
(utveckla gärna i kommentarerna nedan)	2	2.4%

Betyg (reservationer för off-by-one errors)

	3	4	5
Mål	32	18	7
Inluppar	4 (samma för alla)		
Projekt	6 (samma för alla)		
Varav utanför labb	2	2	1

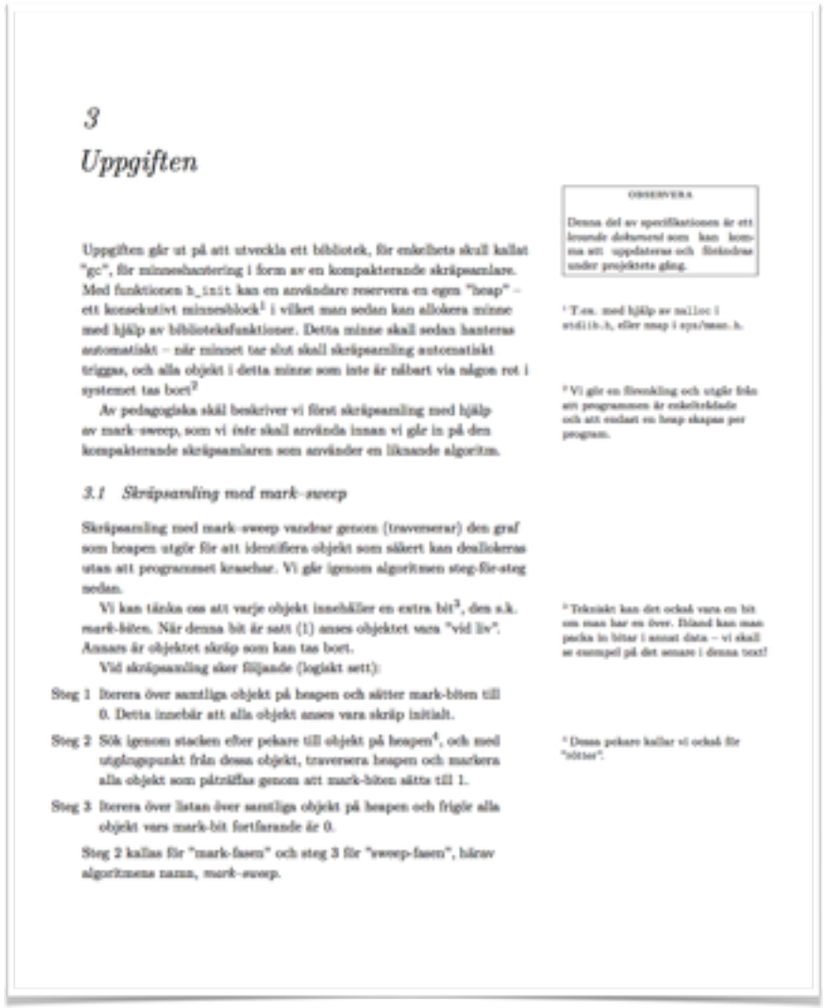
Fas 3: Projektarbete

- Arbeta 4–6 personer (vi tar fram grupperna under november)
- Uppgiften TBA

  Börjar i **december**

  Slutar i **januari** (se kurswebben för detaljer)

- Lämnas in via GitHub
- Presentation och verkstad med annan grupp
- Grupperna lägger själva upp sprintar
- 1–2 KLOC, plus tester



3

Uppgiften

OBSERVERA

Denna del av specifikationen är ett levande dokument som kan komma att uppdateras och förändras under projektets gång.

Uppgiften går ut på att utveckla ett bibliotek, för enkelhets skull kallat "gc", för minneshantering i form av en kompakterande skräpsamlare. Med funktionen `h_init` kan en användare reservera en egen "heap" – ett konsekutivt minnesblock[1] i vilket man sedan kan allokera minne med hjälp av biblioteksfunktioner. Detta minne skall sedan hanteras automatiskt – när minnet tar slut skall skräpsamling automatiskt triggas, och alla objekt i detta minne som inte är nåbart via någon rot i systemet tas bort[2].

Av pedagogiska skäl beskriver vi först skräpsamling med hjälp av mark–sweep, som vi östr skall använda innan vi går in på den kompakterande skräpsamlaren som använder en liknande algoritm.

[1] T.ex. med hjälp av `malloc` i `stdlib.h`, eller `mmap` i `sys/mman.h`.

[2] Vi gör en förenkling och utgår från att programmen är enkeltrådade och att endast en heap skapas per program.

3.1 Skräpsamling med mark–sweep

Skräpsamling med mark–sweep vandrar genom (traverserar) den graf som heapen utgör för att identifiera objekt som säkert kan deallokeras utan att programmet krascher. Vi går igenom algoritmen steg-för-steg nedan.

Vi kan tänka oss att varje objekt innehåller en extra bit[3], den s.k. mark-biten. När denna bit är satt (1) anses objektet vara "vid liv". Annars är objektet skräp som kan tas bort.

Vid skräpsamling sker följande (logiskt sett):

Steg 1 Iterera över samtliga objekt på heapen och sätter mark-biten till 0. Detta innebär att alla objekt anses vara skräp initialt.

Steg 2 Sök igenom stacken efter pekare till objekt på heapen[4], och med utgångspunkt från dessa objekt, traversera heapen och markera alla objekt som påträffas genom att mark-biten sätts till 1.

Steg 3 Iterera över listan över samtliga objekt på heapen och frigör alla objekt vars mark-bit fortfarande är 0.

Steg 2 kallas för "mark-fasen" och steg 3 för "sweep-fasen", härav algoritmens namn, *mark–sweep*.

[3] Tekniskt kan det också vara en bit om man har en över. Ibland kan man packa in bitar i annat data – vi skall se exempel på det senare i denna text!

[4] Dessa pekare kallar vi också för "rötter".

Kodprovet (2x2,5 HP)

- Två frågor — en C, en Java

  Individuellt prov i datorsal, 3 timmar — **ingen tillgång till Internet**

- Syftet: att tvinga alla att sitta i framsätet vid parprogrammering

  Examinerar inte kursmål!

- Går att ta i steg (klara en fråga på varje prov)
- Minst tre provtillfällen under kursen

  24/10 och 11/12 och 19/12

- Anmälan annonseras i Piazza

Simple [minimal sammanfattning]

1. Läs specifikationen och leta specifikt efter **verb** (funktioner/beteende), eller **substantiv** (data/objekt/struktar) — gör en work breakdown structure
2. Skriv kod för att pröva om du tänkt rätt (vad är rätt – hur man kollar det?)
3. Ha alltid ett fungerande program
4. Kompilera efter varje förändring
5. Kör programmet hela tiden för att hitta fel (eller ännu bättre — kör testen!)
6. Dela upp alla problem i delproblem, gå till 7. först när något är enkelt
7. Dela upp alla delproblem i mindre steg — gör de enklaste först
8. **Fuska** (cheat) varje gång du riskerar att fastna
9. **Skarva** (dodge) för att förenkla specifikationer och skapa fler enklare delsteg
10. Växla mellan att: tänka, koda och ibland refaktorera (speciellt **fusk** och **skarvar**)

Simple

- Om du inte redan är en programmerare måste du använda SIMPLE under kursen

- Finns detaljerad beskrivning på kurswebben

  Använder Lab 4, 5 och inlupp 1 som löpande exempel

- Det kan vara svårt att ta in allt direkt, så börja med det som verkar enkelt

  Försök inte göra rätt, utan det som känns rätt — gå tillbaka till texten när det behövs

Att använda en texteditor

- Under kursen kommer vi att använda **Emacs**

  Under fas 2 är de tillåtet att använda IDE:er, men inte rekommenderat

- Emacs kan också betyda vim men **inte**

  Gedit

  Nano

  Notepad++

  Sublime

  …

Sammanfattningsvis

- Från och med nästa vecka skall du jobba med inlämningsuppgifterna

  Jobba i ett programmeringspar

- Labbarna är till för redovisning och hjälp och är **inte obligatoriska**
- Kursen kretsar till 75% kring redovisning av **mål** som finns beskrivna på kursens webbsida
- Alla mål redovisas i par men betygsätts individuellt
- Du förväntas själv göra kopplingen mellan mål och uppgifter

  Viss handledning finns i form av tips i uppgiftstexterna

- **Implementera först, redovisa sedan**