Föreläsning 9

Tobias Wrigstad

”Hur man tänker” — design & process

”Hur man tänker”

Att tänka i lager

Närmare domänen

Närmare maskinen

`ask_string_question` — *Ställ fråga, läs strängsvar*

`getline` — *Läs en rad*

`getchar` — *Läs ett tecken*

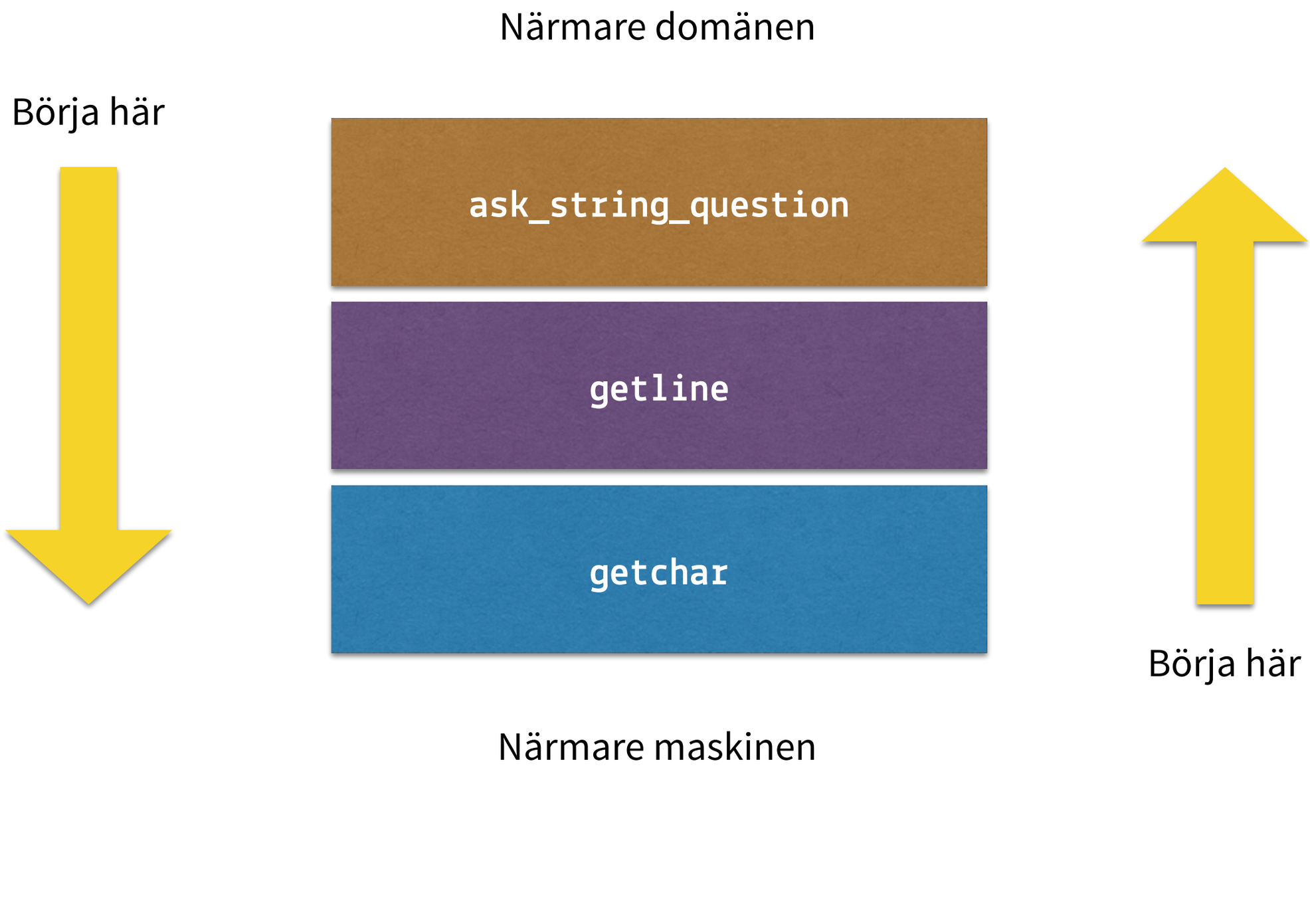
Att tänka i lager

För varje funktion/strukt — vilket lager tillhör den?

Vad bör den (inte) vara beroende av?

Vilka lager ”behöver” jag?

Top-down eller bottom up?



Att arbeta bottom-up

- Börja på den ”lägsta nivån”

  Högre teknisk komplexitet, ofta helt orelaterade till domänen

- Nackdelar/risker:

  Att du gör något som inte behövs senare

  Det kan kännas motigt att vara så långt borta från domänen/specifikationen

- Konsekvenser:

  Mindre fuskande eftersom varje funktion bygger på de som vi just implementerat

Att arbeta top-down

- Börja på den ”högsta nivån”

  Omfattar i regel mest logik som hör till specifikationen/domänen

- Nackdelar/risker:

  Man kan fatta dumma designbeslut på grund av tekniska omständigheter som man inte känner till ännu

  Kan vara svårt att ha ett körande program hela tiden

- Konsekvenser:

  Ofta mer fuskande eftersom man ”knuffar funktionalitet framför sig”

Vilket skall jag välja?

- Det är litet beroende på person

  Om du inte direkt vet vilket som är bäst för dig — prova båda och utvärdera!

- **Min rekommendation**: top-down är bättre för den som känner sig osäker på C

Implementation av frågor i lagerhanteraren

Top Down

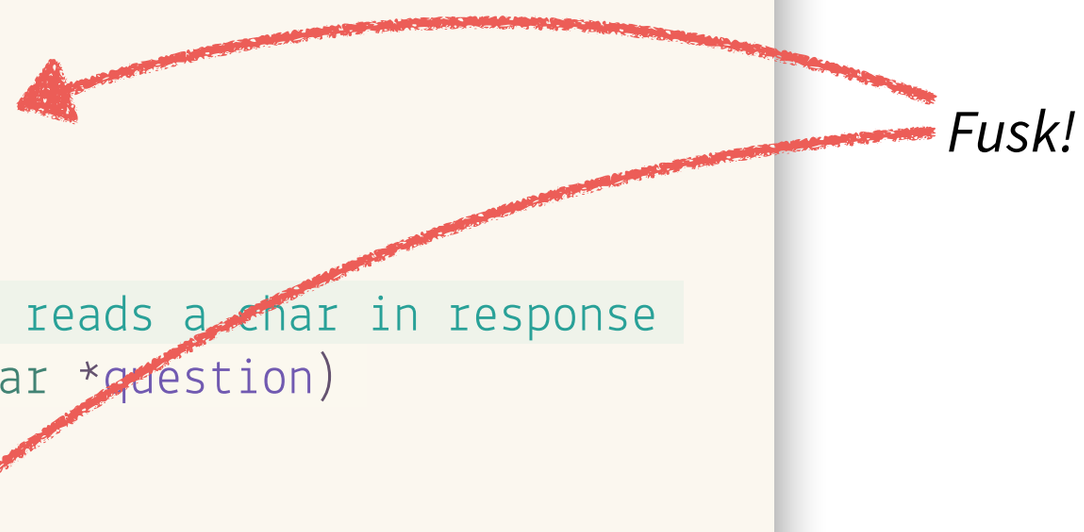
```
/// Asks a question and reads a string in response
char *ask_string_question(char *question)
{
  puts(question);
  return read_string();
}
```

```
/// Asks a question and reads a char in response
int ask_int_question(char *question)
{
  puts(question);
  return read_int();
}
```

Top Down

Implementation av frågor i lagerhanteraren

```c
/// Asks a question and reads a string in response
char *ask_string_question(char *question)
{
  puts(question);
  return read_string();
}

/// Asks a question and reads a char in response
int ask_int_question(char *question)
{
  puts(question);
  return read_int();
}
```



Top Down

Implementation av frågor i lagerhanteraren

Löser fusk 2 från föregående bild!

```c
/// Reads a line from the keyboard and converts it to an int
int read_int()
{
  char *buf = read_string();
  int result = atol(buf);

  free(buf);
  return result;
}
```

Fusk!

***Skarv**: anta att input alltid är valida heltal*

Top Down

Implementation av frågor i lagerhanteraren

Löser resterande fusk från föregående bilder!

```c
/// Reads a line from the keyboard, puts it on the heap and returns a
/// pointer to it
char *read_string()
{
  char *buf = NULL;
  size_t len = 0;
  ssize_t read = getline(&buf, &len, stdin);
  buf[read-1] = '\0'; // Skarv 1 & 2
  return buf;
}
```

***Skarv 1**: anta att vi aldrig vill ha newline kvar!*

***Skarv 2**: anta att newline == ’\n’ (stämmer ej på Windows)*

Bottom up

Implementation av frågor, bottom-up

- I stort sett som top down, fast baklänges

  Först `read_string`

  Sedan `read_int` ovanpå `read_string`

  Sedan `ask_int_question` ovanpå `read_int`

- Det är ett visst avstånd som måste överbryggas från vad programmet vill (`ask_int_question`) och `read_string`.

  Gör på det sätt du själv känner att det är lättast att tänka

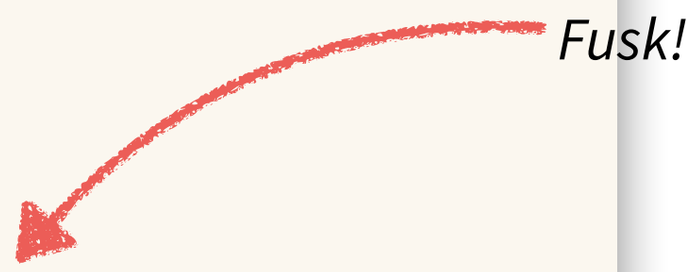
Nästa steg: ta bort skarvarna

```c
/// Reads a line from the keyboard, puts it on the heap and returns a
/// pointer to it
char *read_string(bool strip_newline)
{
  char *buf = NULL;
  size_t len = 0;
  ssize_t read = getline(&buf, &len, stdin);
  if (strip_newline && read > 0) buf[read-1] = '\0'; // Skarv 2 är kvar
  return buf;
}
```

Användaren får välja om strängen skall ha kvar newline!

Nästa steg: ta bort skarvarna

```
/// Reads a line from the keyboard, puts it on the heap and returns a
/// pointer to it
int read_int(bool repeat_until_valid_int)
{
  char *buf = NULL;

  do {
      if (buf) free(buf);

      buf = read_string(true);

  } while (repeat_until_valid_int && is_number(buf) == false);

  int result = atol(buf);
  free(buf);
  return result;
}
```



Nästa steg: ta bort skarvarna

”Ha alltid ett körbart program”

```c
/// Returns true if a string only has digits
bool is_number(char *str)
{
  return true;
}
```

Gör så att vi kan testa programmet, men funkar förstås inte för icke-valid input!

Nästa steg: ta bort skarvarna

Löser fusket från föregående bild!

```
/// Returns true if a string only has digits
bool is_number(char *str)
{
  bool valid_int = true;

  for (char *c = str; *c && valid_int; ++c)
    {
      valid_int = isdigit(*c);
    }

  return valid_int;
}
```

Loopa igenom strängen och kolla att varje char är en siffra

`read_string` lägger en sträng på heapen

- Inte alltid rätt, t.ex. i `read_int`
- Tänk om jag vill läsa in direkt i en `char`-array i databasen?
- Logiken är densamma oavsett var i minnet man läser in strängen

  Bra idé: bryt ut detta ur funktionen så blir den mer generell

  Lättare att återanvända

  Lättare att testa

- Sedan kan vi bygga en ekvivalent `read_string` ovanpå den generella, som sparar en sträng på heapen

Nästa steg: ta bort skarvarna

En mer generell funktion för att läsa in strängar!

```c
/// Reads a line from the keyboard, puts it in the len-sized
/// memory space pointed to by buf, and optionally removed newlines
char *read_string_with_buffer(char *buf, size_t len, bool strip_newline)
{
  ssize_t read = getline(&buf, &len, stdin);
  if (read > 0 && strip_newline)
    {
      buf[read-1] = '\0'; // -2  på Windows…
    }
  return buf;
}
```

***Skarv**: utgår från att vi inte kör på Windows…*

Undvik onödig upprepning

Vi kan enkelt implementera om read_string() i termer av read_string_with_buffer()

```
/// Reads a line from the keyboard, removes newlines,
/// puts on the heap and returns a pointer to it
char *read_string()
{
  return read_string_with_buffer(NULL, 0, true);
}
```

Oftast behöver man inte newline — vill man ha det får man använda `read_string_with_buffer`

Inga magiska konstanter (läsbarhet)

```
/// Uses the improved read_string_with_buffer
int read_int(bool repeat_until_valid_int)
{
  char *buf = alloca(16); // 16 is a big number
  int len = 16;

  do
    {

      buf = read_string_with_buffer(buf, len, true);

    } while (repeat_until_valid_int && is_valid_int(buf) == false);

  return atol(buf);
}
```

?!

Inga magiska konstanter (läsbarhet)

```
#define STRIP_NEWLINE true

/// Uses the improved read_string_with_buffer
int read_int(bool repeat_until_valid_int)
{
  char *buf = alloca(16); // 16 is a big number
  int len = 16;

  do {

    buf = read_string_with_buffer(buf, len, STRIP_NEWLINE);

  } while (repeat_until_valid_int && is_valid_int(buf) == false);

  return atol(buf);
}
```

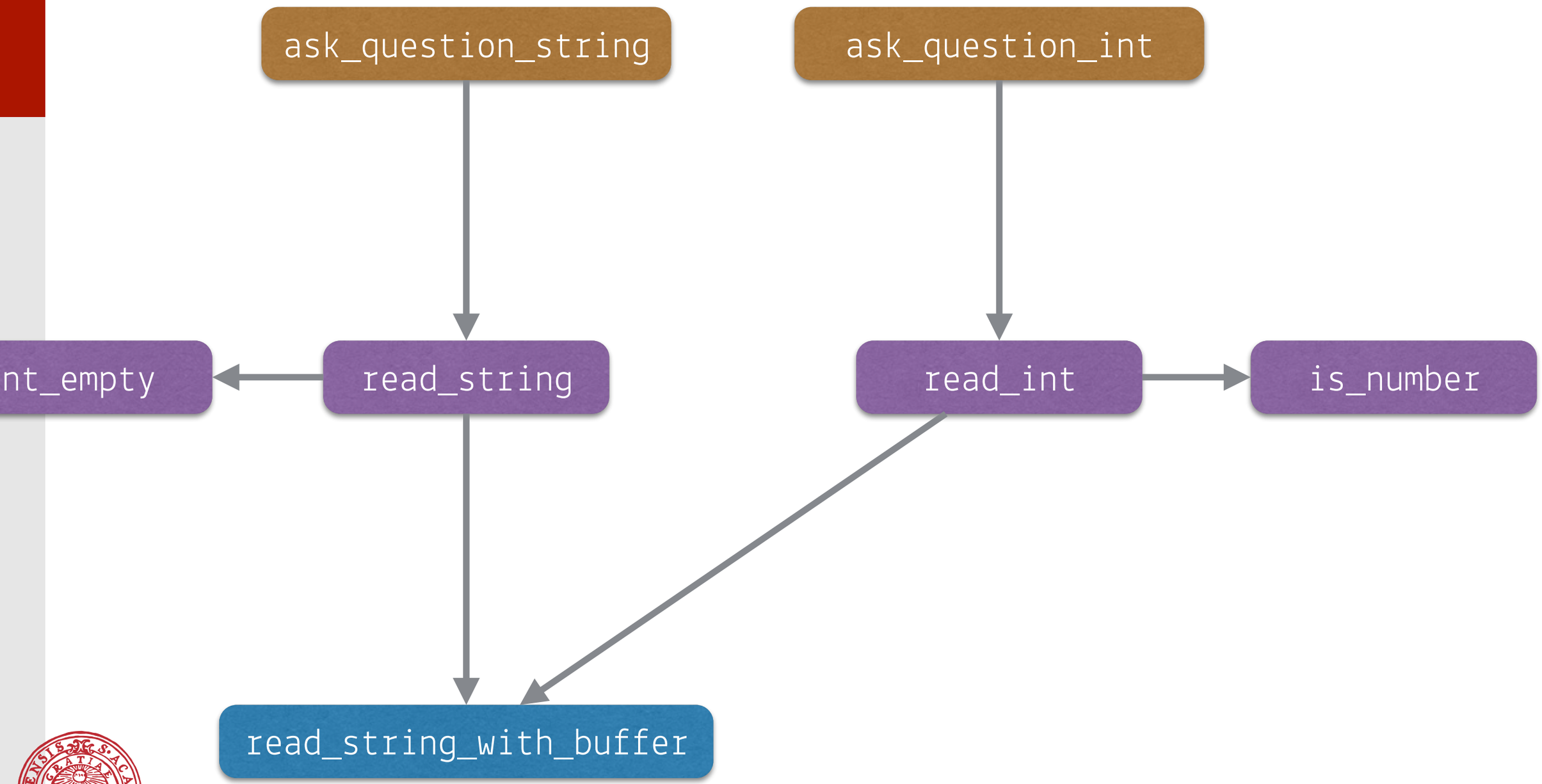
Inga magiska konstanter (läsbarhet)

Vi kan enkelt implementera om read_string() i termer av read_string_with_buffer()

```c
/// Reads a line from the keyboard, removes newlines,
/// puts on the heap and returns a pointer to it
char *read_string()
{
  return read_string_with_buffer(NULL, 0, STRIP_NEWLINE);
}
```

Oftast behöver man inte newline — vill man ha det får man använda `read_string_with_buffer`

Beroenden


ask_question_string
ask_question_int
nt_empty
read_string
read_int
is_number
read_string_with_buffer

Ytterligare påbyggnad

- Med hjälp av våra två ask kan vi enkelt bygga en ask shelf (t.ex. A25)

```
/// Reads the necessary data for a shelf_t, constructs a
/// shelf_t, and returns it
shelt_t ask_shelf_question()
{
  shelf_t shelf;
  shelf.name   = ask_string_question("Mata in ett tecken”)[0];
  shelf.number = ask_int_question("Mata in ett tal”);
  return shelf;
}
```

***Skarv**: förutsätter valitt indata*

Fixa skarven: validera

```
shelf_t ask_shelf_question()
{
  shelf_t shelf; // har char name; int number;

  char *name = NULL;
  do {
    if (name) free(name);
    name = ask_string_question("Mata in ett tecken");
    shelf.name = name[0];
  } while (strlen(name != 1));

  long number = 0;
  do {
    number = ask_int_question("Mata in ett tal 0--99");
    shelf.number = number;
  } while (0 <= number && number < 100);

  return shelf;
}
```

Repetition!

”Bad smell: upprepning”

- Det finns ett mönster som upprepas

  Läs in data

  Validera

  (Ta bort temporära data)

  Skapa efterfrågad struktur

- Samma mönster, men olika beteende för olika data

```c
shelf_t ask_shelf_question()
{
  shelf_t shelf; // har char name; int number;

  char *name = NULL;
  do {
    if (name) free(name);
    name = ask_string_question("Mata in ett tecken");
    shelf.name = name[0];
  } while (strlen(name != 1));

  long number = 0;
  do {
    number = ask_int_question("Mata in ett tal 0--99");
    shelf.number = number;
  } while (0 <= number && number < 100);

  return shelf;
}
```

Generalisering

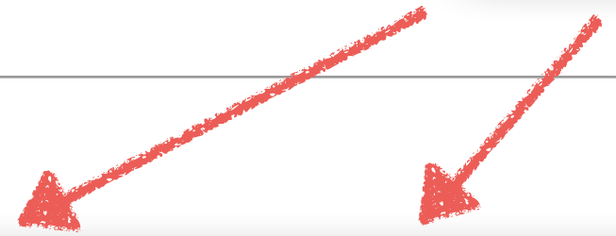
- Kan vi skapa en funktion som följer mönstret men som gör rätt sak för rätt data?
- Försök ett: vi skickar in ”flaggor” för att tala om vad för data etc. skall läsas in
  - + Löser problemet
  - – Koden blir väldigt komplicerad
  - – Går inte att utöka för data som vi inte känner till
- Försök två: bryt ut beteende med hjälp av funktionspekare
  - + Löser problemet
  - + Framtidssäker eftersom logiken tillhandahålls av användaren

Funktionspekare

```
/// Valideringsfunktion tar emot en pekare till
/// data och kontrollerar om datat kan omvandlas
/// till avsedd typ
/// Exempel:
///   - valid_int
///   - valid_shelf
typedef bool  (*v_f)(char *);

/// Konstruktor som tar emot en sträng med validerat
/// data och omvandlar det till avsedd typ, returnerar
/// en pekare till det nya datat
/// Exempel:
///   - str_to_int
///   - str_to_shelf
typedef void *(*m_f)(char *);
```

Generaliserad fråga

```c
typedef bool  (*v_f)(char *);
typedef void *(*m_f)(char *);
```



```c
void *ask_question(char *q, v_f validate, m_f convert, bool cleanup)
{
  // Ask question until optional validation of input is satisfied
  char *input = NULL;
  do {
    puts(q);
    if (input) free(input);
    input = read_string();
  } while (validate && validate(input) == false);

  // If a conversion function was specificied, convert input
  void *result = convert ? convert(input) : input;

  if (cleanup) free(input);
  return result;
}
```

Validera data, skapa data

```
bool valid_shelf(char *input)
{
  return strlen(input) == 3 && isalpha(input[0]) && valid_int(input+1);
}

shelf_t *str_to_shelf(char *input)
{
  shelf_t *shelf = malloc(sizeof(shelf_t));
  shelf->name = input[0];
  shelf->number = atol(input+1);
  return shelf;
}
```

Slutlig ask_shelf/string_question

```
shelf_t ask_shelf_question()
{
  return *ask_question("Mata in en hyllplats (tecken, följt av siffra 0-99)",
                       valid_shelf,
                       str_to_shelf,
                       true);
}
```

```
char *ask_string_question()
{
  return *ask_question("Mata in en sträng",
                       NULL,
                       NULL,
                       false);
}
```

Skarv: *fungerar för heltal, men "fult"*

Förbättring: unioner

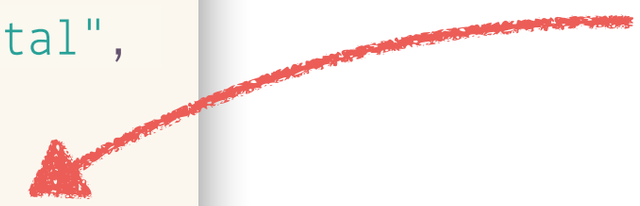
```c
// ändra void * => result_t i ask_question och m_f
typedef union result result_t;

union result
{
  void *ptr;
  long int_value;
  char char_value;
};

result_t str_to_shelf(char *input)
{
  shelf_t *shelf = malloc(sizeof(shelf_t));
  shelf->name = input[0];
  shelf->number = atol(input+1);
  return (result_t) { .ptr = shelf };
};
```

Lösning med hjälp av unioner (nästan samma)

```c
result_t str_to_int(char *s)
{
  return (result_t) { .int_value = atol(s) };
}
```

```c
result_t str_to_str(char *s)
{
  return (result_t) { .ptr = s };
}
```

```c
int ask_int_question()
{
  return ask_question("Mata in ett heltal",
                      valid_int,
                      str_to_int,
                      true).int_value;
}
```



```c
x = foo()
return x.bar;
```

är samma som

```c
return foo().bar;
```

Till slut

```
shelf_t *ask_shelf_question()
{
  return ask_question("Mata in en hyllplats (tecken, följt av siffra 0-99)",
                      ok_shelf,
                      make_shelf,
                      true).ptr;
}
```

```
char *ask_string_question()
{
  return ask_question(
    "Mata in en sträng",
    NULL,
    str_to_str,
    false).ptr;
}
```

```
int ask_int_question()
{
  return ask_question(
    "Mata in ett heltal",
    valid_int,
    str_to_int,
    true).int_value;
}
```

Ännu bättre: exponering för programmet med hjälp av makron

```
/// Grundläggande funktioner
#define Ask_int(q)          ask_question(q, valid_int, str_to_int, true)
#define Ask_str(q)          ask_question(q, NULL,      str_to_str, false)

/// Funktion för specifika återkommande frågor
#define Ask_namn()          Ask_str("Namn:")
#define Ask_beskrivning( )  Ask_str("Beskrivning:")
#define Ask_pris()          Ask_int("Pris:")
#define Ask_lagerhylla()    ask_question("...", valid_shelf, str_to_shelf, true)
#define Ask_antal()         Ask_int("Antal:")
```

OBS! Detta kan man alltså göra även utan funktionspekare och unioner!

Titta nu på add_goods — hur ”ren” den blir

```
void add_goods(db_t *db)
{
  goods_t g;

  g.name   = Ask_namn();
  g.desc   = Ask_beskrivning();
  g.price  = Ask_pris();
  g.shelf  = Ask_lagerhylla();
  g.amount = Ask_antal();

  db->goods[db->total] = g;
  ++db->total;
}
```

***Skarv**: följer inte specen (inget val spara/redigera)*

Ta bort skarven — fortfarande snyggt

```
void add_goods(db_t *db)
{
  goods_t g;

  do {
    g.name   = Ask_namn();
    g.desc   = Ask_beskrivning();
    g.price  = Ask_pris();
    g.shelf  = Ask_lagerhylla();
    g.amount = Ask_antal();

    char answer = Ask_char("Spara? (ja/nej)");
  } while (strchr("Jj", answer) == false);

  db->goods[db->total] = g;
  ++db->total;
}
```

Fusk!

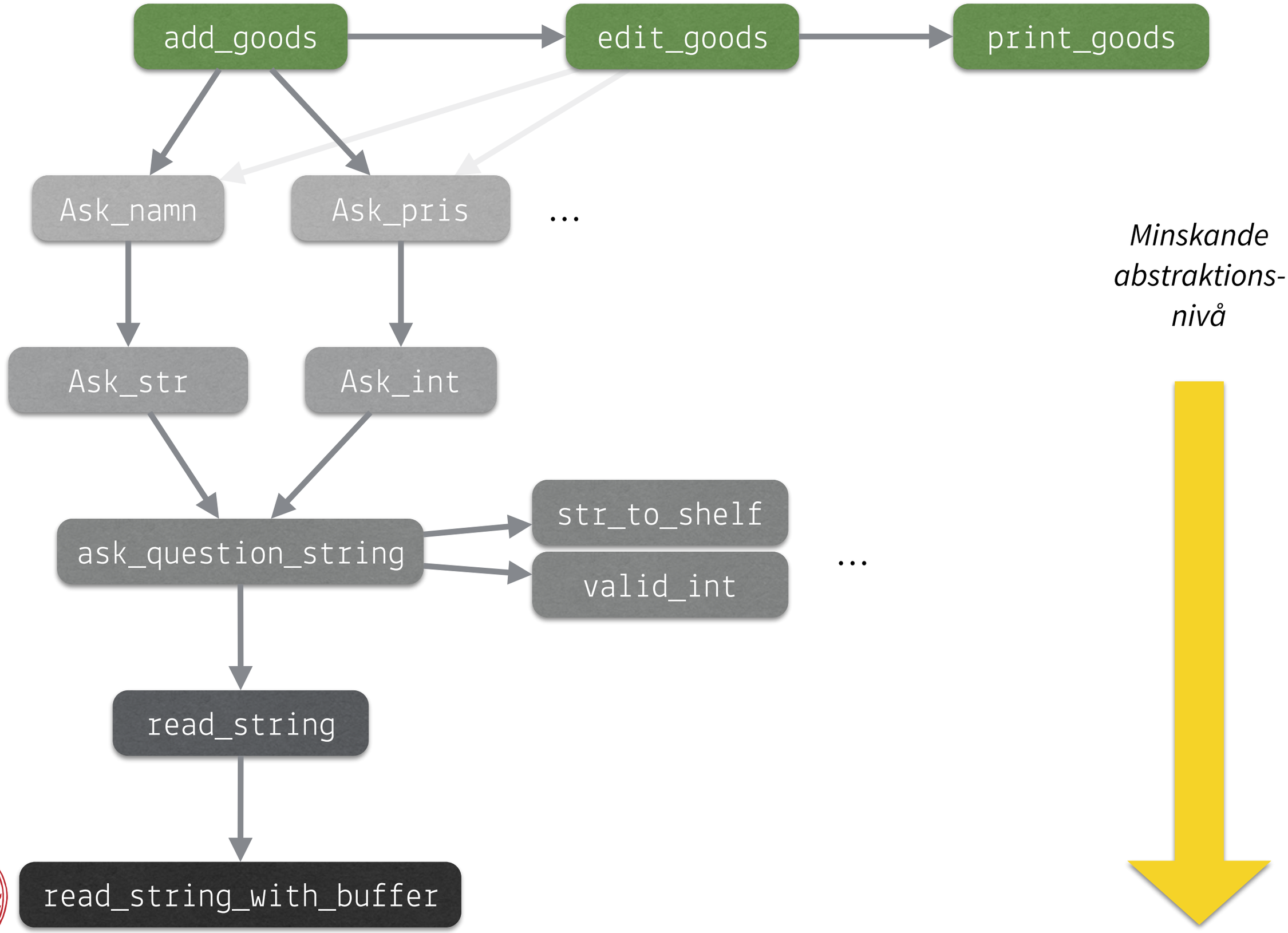
***Skarv**: följer inte specen (inget redigera-val)*

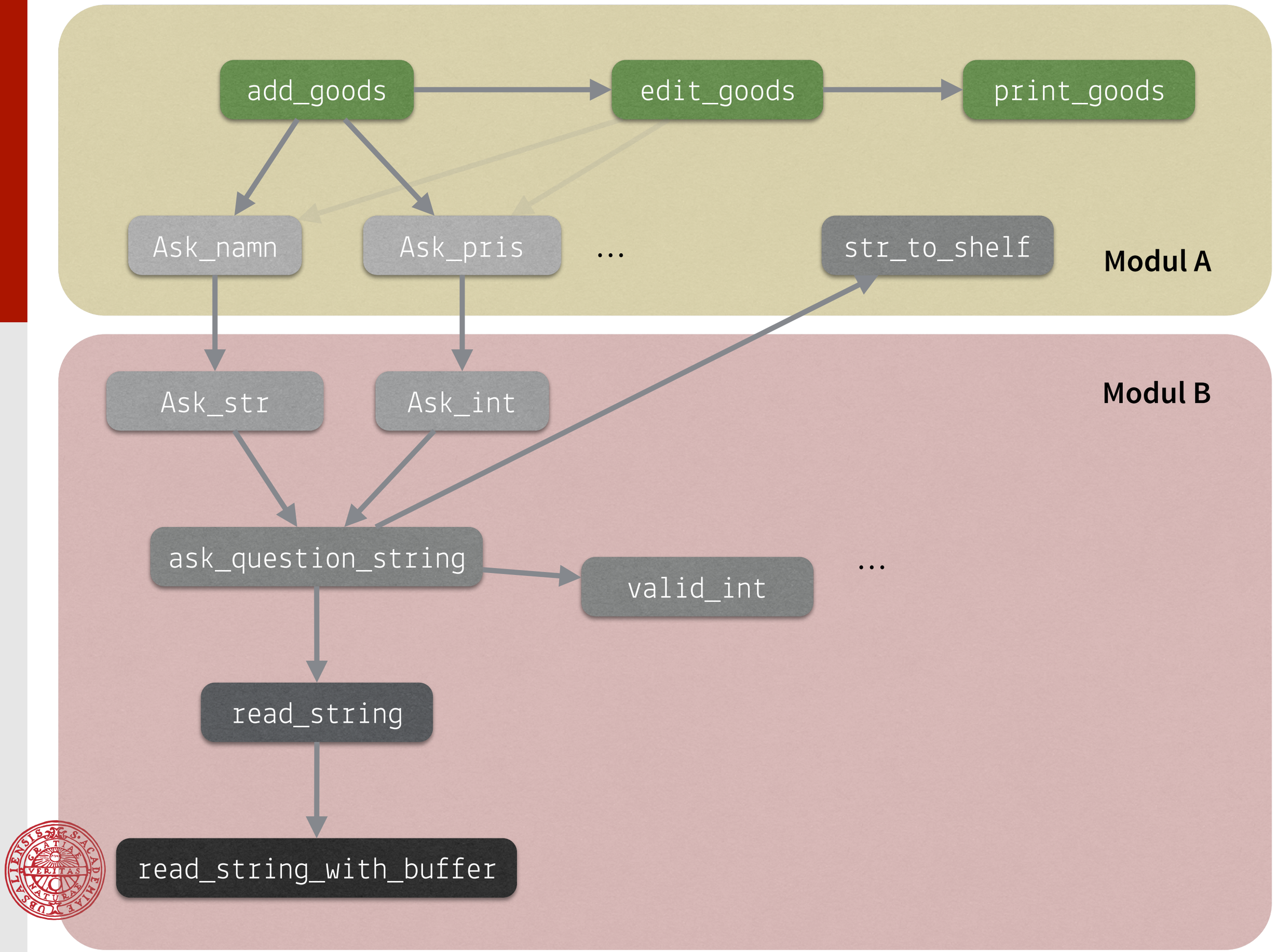
Återanvändning i `edit_goods`

```
void edit_goods(db_t *db, goods_t *g)
{
  char answer = Ask_char();

  goodt_g copy = *g;

  switch (answer) {
  case 'N':
  case 'n': copy.name = Ask_namn(); break;
   // etc.
  case 'P':
  case 'p': copy.price = Ask_pris(); break;
 }

  print_goods(copy);  // fusk!

  char answer = Ask_char("Spara? (ja/nej)");
  if (strchr("Jj", answer) == false)
    {
      *g = copy;
    }
}
```

```
"Vad vill du redigera?\n"
"[N]amn\n"
// etc
"[P]ris"
```

Kanske skulle detta ges en egen funktion som också anropades i add_goods?


add_goods
edit_goods
print_goods
Ask_namn
Ask_pris
...
Ask_str
Ask_int
ask_question_string
str_to_shelf
valid_int
...
read_string
read_string_with_buffer
Minskande abstraktions-nivå


add_goods
edit_goods
print_goods
Ask_namn
Ask_pris
...
str_to_shelf
Modul A
Ask_str
Ask_int
Modul B
ask_question_string
valid_int
...
read_string
read_string_with_buffer

Krångligare med så många mellansteg?

- Se på koden för `add_goods`, den är **tydlig** eftersom den slipper bry sig så mycket om tekniska detaljer

  Procedurell abstraktion: tydligt vad varje funktion gör, även utan insyn

  Endast ett anrop till `getline` i all kod

- I många andra funktioner, t.ex. edit, kunde jag återanvända `Ask_`-funktionerna och därigenom få lika fin och ren kod som i `add_goods` — ”gratis”

- Observera att man måste inte ha ”supergenerella” funktioner i botten

  Man kan ha separata `read`-funktioner utan funktionspekare etc.

Sammanfattning

- Top-down eller bottom-up

  Vad är rätt för dig?

- Lagertänkande

  Bygger abstraktioner bit för bit

    Lager är **inte** detsamma som moduler

- Generella byggstenar kan återanvändas
- Programmera nära domänen
- Göm tekniska komplexiteter ”längre ned”

Läsbar kod

- Notera att den längsta funktionen här är ~20 rader (`edit_goods`) — den är för lång!
- De flesta funktionerna är ca 5–6 rader — en bra längd
- Funktioner skall helst bara göra en sak
- Om de har för många rader så blir det svårt att överblicka vad de gör

Svårt att se att de är korrekta

Svårt att underhålla, förstå, etc.

Föreläsning 10

Abstraktion, modularisering och informationsgömning

… separatkompilering & headerfiler

Abstraktion

- Tänkandet i abstraktioner är ett grundläggande mänskligt drag sedan ca 100.000 år
- Att tänka bort vissa egenskaper hos ett föremål eller en företeelse och därigenom lyfta fram andra. Hur man väljer beror på vilket syfte man har med abstraktionen.

  En modell är med nödvändighet en abstraktion

- Många instanser ligger till grund för en abstraktion som beskriver och grupperar instanserna och gör det lättare att resonera om individerna

  Kraftfullt verktyg, jämför t.ex. facktermer

Kontrollabstraktion

- Ett program är uppdelat i subrutiner som anropas och returnerar till anroparen

  Hur kontrollen flödar i ett program blir väsentligt förenklat

  Stackmekanismen (läs kompendium i kursens repo och stack och heap!) stöder denna grundläggande abstraktion

  Subrutiner utan returvärde – procedurer; med returvärde – funktioner

- Undantagshantering är en annan kontrollabstraktion som vi skall se senare
- Inlining – undviker litet av overheaden av kontrollabstraktion
- Procedurabstraktion

  Vi kan skilja mellan funktionens specifikation och dess implementation i termer av mer primitiva funktioner

(nästan alltid irrelevant)

Lorem ipsum dolor
sit amet, consectetuer
    adipiscing elit.
Praesent
    lacinia
    tellus ac
orci
    laoreet
    tincidunt.
Phasellus dictum

fermentum lacus.
Vestibulum quam.
Duis porta,
nibh
sed
    congue
tincidunt,
    risus
        felis
        porttitor orci,
        vitae pharetra erat arcu eu
    dolor.
        Sed lacus nulla, auctor eget,
interdum eget, tempus sed,
pede.

Ut odio mauris,
tincidunt ac,
molestie eu,
sagittis at,
wisi.
    Ut
    porttitor, est
    eget accumsan
semper, neque
turpis dictum
    quam,

adressFromContacts

Ut odio mauris,
tincidunt ac,
molestie eu,
sagittis at,
wisi.
    Ut
    porttitor, est
    eget accumsan
semper, neque
turpis dictum
    quam,

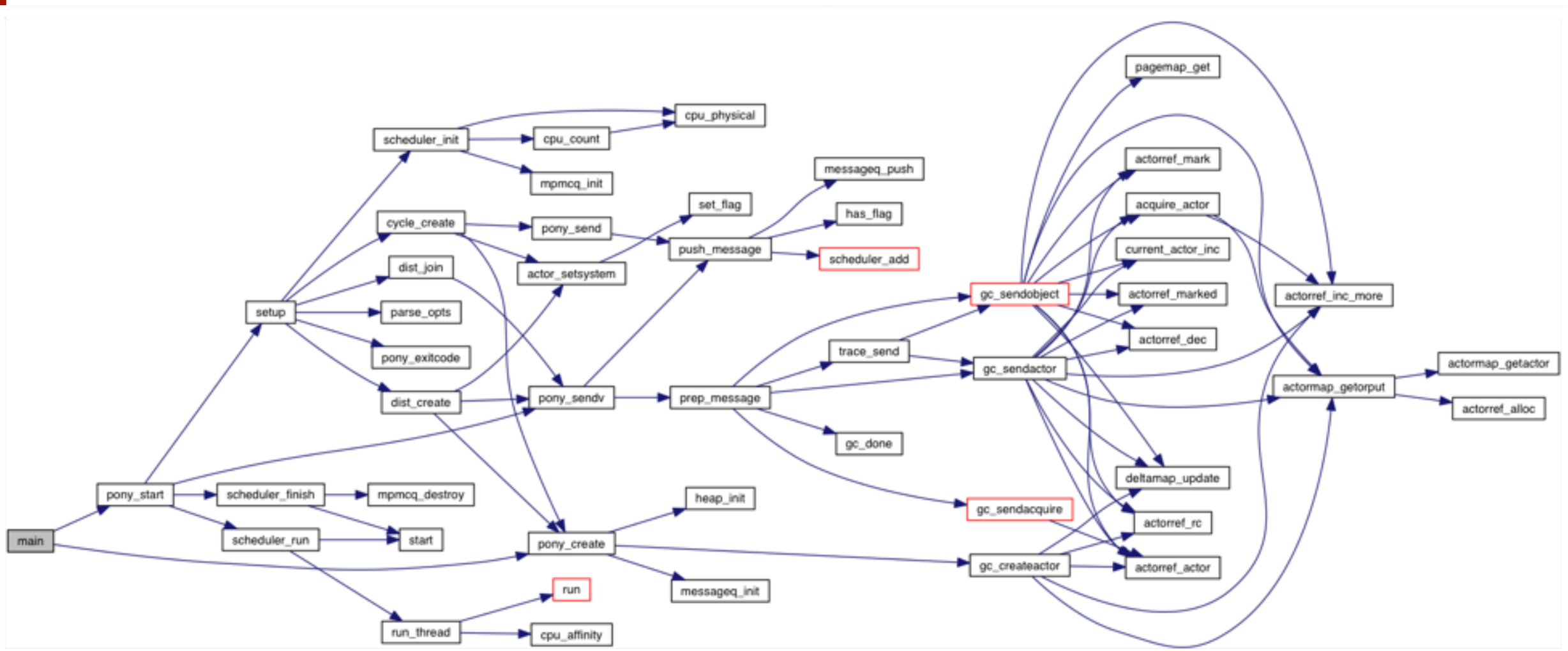
sendMail

fermentum lacus.
Vestibulum quam.
Duis porta,
nibh
sed
    congue
tincidunt,
    risus
        felis
        porttitor orci,
        vitae pharetra erat arcu eu
    dolor.
        Sed lacus nulla, auctor eget,
interdum eget, tempus sed,
pede.

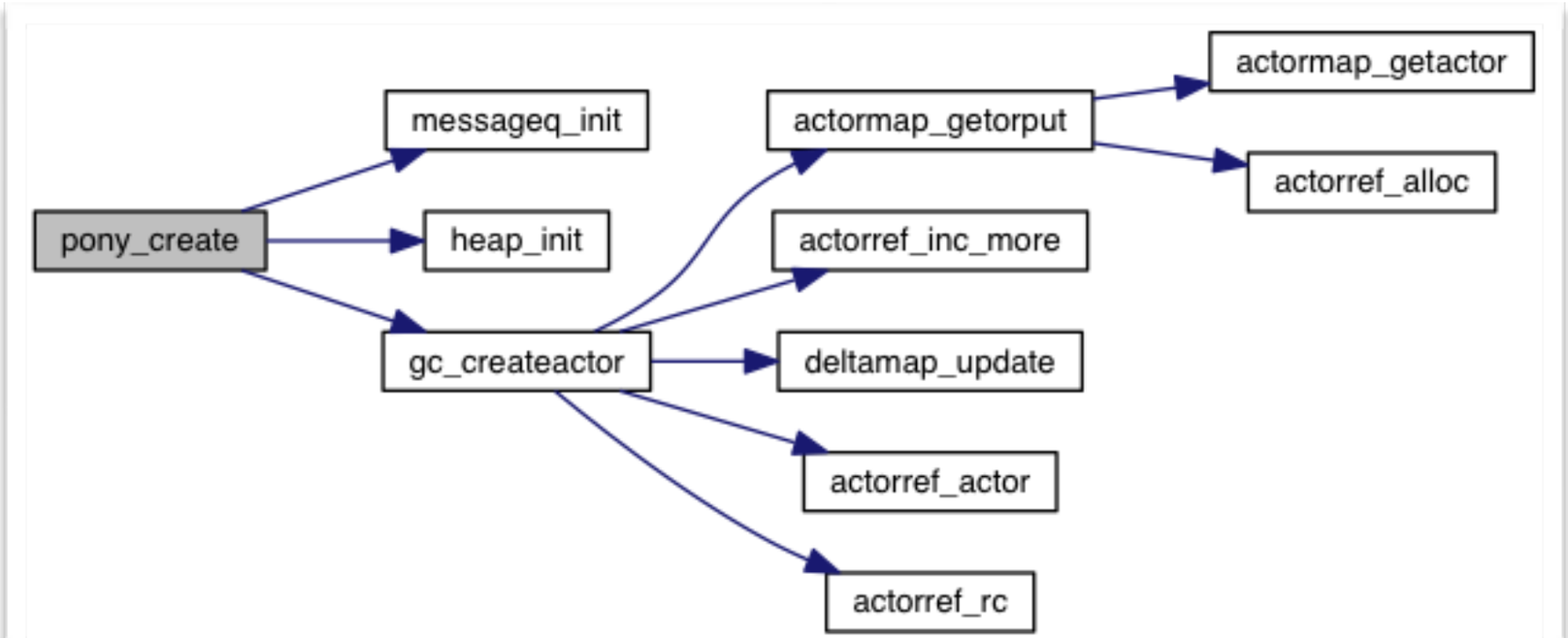
saveToSentFolder

Lorem ipsum dolor
sit amet, consectetuer
    adipiscing elit.
Praesent
    lacinia
    tellus ac
orci
    laoreet
    tincidunt.
Phasellus dictum

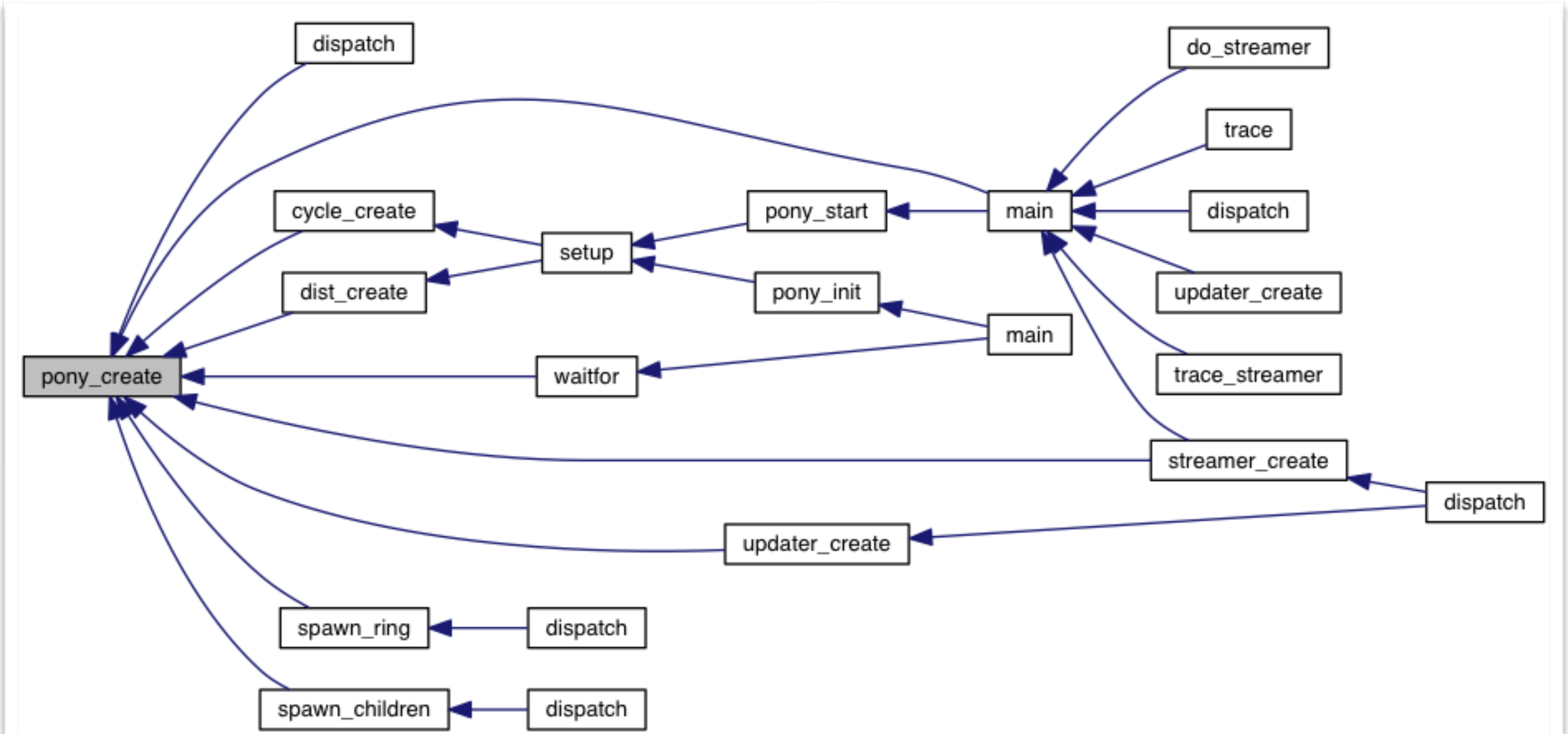
Anropsgraf



Doxygen (”vårt” dokumentationsverktyg för C) kan generera statiska anropsgrafer



Anropsgraf rotad i `pony_create`



varifrån anropas `pony_create`?

Dataabstraktion

- Strukturera programmen så att de använder/manipulerar ”abstrakta data”

  Programmen bör inte förutsätta att datat är implementerat på ett särskilt vis

  Programmen bör inte förutsätta att datat har andra egenskaper än de som är nödvändiga för att utföra den aktuella uppgiften

- Abstrakta data: ”specifikation”
- Konkret data: den faktiska implementationen som idealiskt är oberoende av alla program som använder datat

  Använd accessorer och mutatorer för manipulera datat som blir bryggan mellan det abstrakta och konkreta datat

  Kapsla in implementationsdetaljer och undvik externa beroenden

Dataabstraktion

- Strukturera programmen så att de använder/manipulerar ”abstrakta data”

  Programmen bör inte förutsätta att datat är implementerat på ett särskilt vis

  Programmen bör inte förutsätta att datat har andra egenskaper än de som är nödvändiga för att utföra den aktuella uppgiften

- Abstrakta data: ”specifikation”

- Konkret data: den faktiska implementationen som idealiskt är oberoend program som använder datat

  Använd accessorer och mutatorer för manipulera datat som blir brygg abstrakta och konkreta datat

  Kapsla in implementationsdetaljer och undvik externa beroenden

En int är 64 bitar

Pris representeras som ett flyttal

Ett personnummer är en sträng

Dataabstraktion

- Strukturera programmen så att de använder/manipulerar ”abstrakta data”

  Programmen bör inte förutsätta att datat är implementerat på ett särskilt vis

  Programmen bör inte förutsätta att datat har andra egenskaper än de som är nödvändiga för att utföra den aktuella uppgiften

- Abstrakta data: ”specifikation”
- Konkret data: den faktiska implementationen som idealiskt är oberoende av alla program som använder datat

  Använd accessorer och mutatorer för manipulera datat som blir bryggan mellan det abstrakta och konkreta datat

  Kapsla in implementationsdetaljer och undvik externa beroenden

Kopplar loss en datastrukturs klienter från datastrukturen!

Dataabstraktion

- Strukturera programmen så att de använder/manipulerar ”abstrakta data”

  Programmen bör i… …å ett särskilt vis

  Programmen bör i… …per än de som är nödvändiga för att utföra den aktuella uppgiften

```
commodity_t book = ... ;
book.cost = 12.50;
int cost_of_book = book.cost;
```

- Abstrakta data: ”specifikation”
- Konkret data: den faktiska implementationen som idealiskt är oberoende av alla program som använder datat

  Använd accessorer och mutatorer för manipulera datat som blir bryggan mellan det abstrakta och konkreta datat

  Kapsla in implementationsdetaljer och undvik externa beroenden

Dataabstraktion

- Strukturera programmen så att de använder/manipulerar ”abstrakta data”

  Programmen bör i… …på ett särskilt vis

  Programmen bör i… …ber än de som är nödvändiga för att utföra den aktuella uppgiften

```
commodity_t book = ... ;
book.cost = 12.50;
int cost_of_book = book.cost;
```

- Abstrakta data: ”specifikation”

- Konkre… …alla progra…

```
int cost_in_eurocent(commodity_t c);
void set_cost_in_eurocent(commodity_t *c, int cent);
```

  Använd accessorer och mutatorer för manipulera datat som blir bryggan mellan det abstrakta och konkreta datat

  Kapsla in implementationsdetaljer och undvik externa beroenden

Effekten och vikten av abstraktion

- Höga abstraktioner

  + förbättrar läsbarheten och överskådligheten

  + underlättar utveckling (flexibilitet, förändring)

  – tenderar att försämra prestanda något (varför?!)

(nästan alltid irrelevant)

- Designprincip:

  Använd god kontroll- och dataabstraktion alltid

  Eventuella undantag måste upptäckas den hårda vägen, aldrig via spekulation

Modularisering

- En designprincip — program delas in i moduler (jmf. ett monolitiskt system med en modul)

  Varje modul är ett ”delprogram” med ansvar för specifika åtaganden

  En modul behöver inte vara programspecifk (jmf. t.ex. `stdlib` i C; eller en lista)

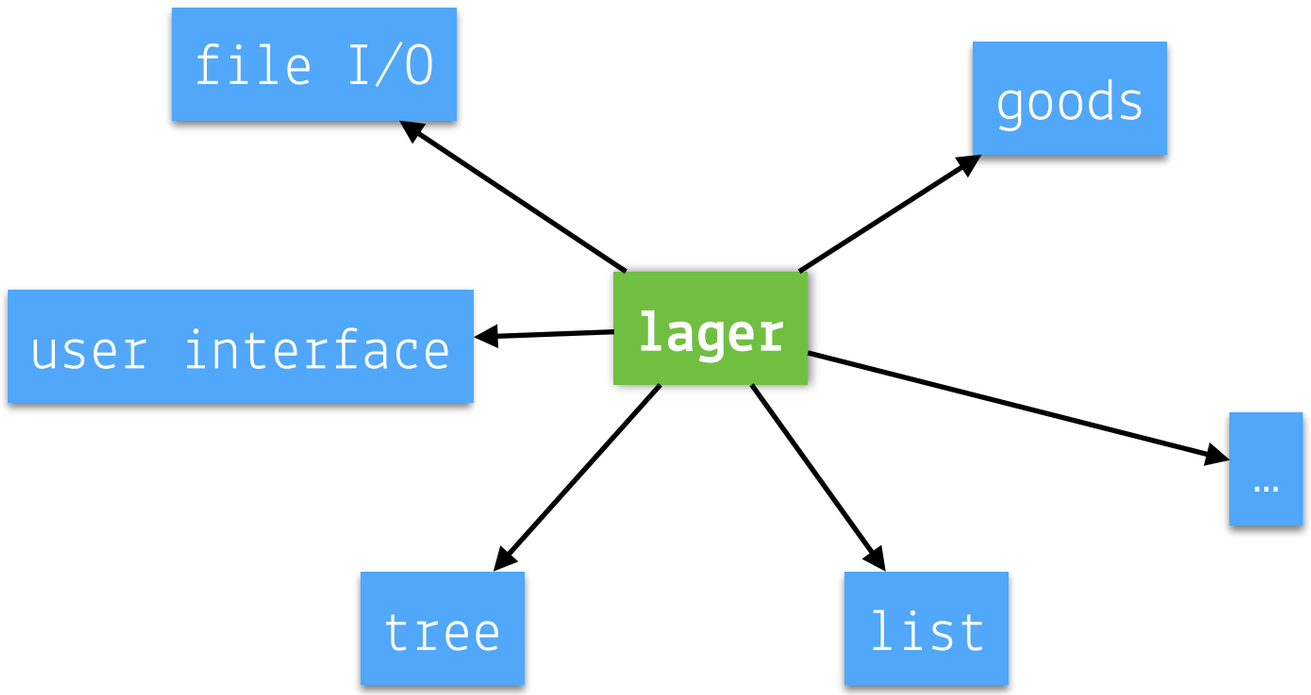
`lager`

Modularisering

- En designprincip — program delas in i moduler (jmf. ett monolitiskt system med en modul)

  Varje modul är ett ”delprogram” med ansvar för specifika åtaganden

  En modul behöver inte vara programspecifk (jmf. t.ex. `stdlib` i C; eller en lista)



Fördelar med modularisering

- *Många små enheter* är enklare än *få större* att…
  - överblicka,
  - navigera, och
  - återanvända
- En design i flera moduler möjliggör parallell utveckling
- En moduls funktioner och data kan kapslas in
  - Förenklar återanvändning
  - Skyddar mot propagerande förändringar
  - Förbättrad underhållsbarhet

Lorem ipsum dolor
sit amet, consectetuer
adipiscing elit.
Praesent
lacinia
tellus ac
orci
laoreet
tincidunt.
Phasellus dictum

fermentum lacus.
Vestibulum quam.
Duis porta,
nibh
sed
congue
tincidunt,
risus
felis
porttitor orci,
vitae pharetra erat arcu eu
dolor.
Sed lacus nulla, auctor eget,
interdum eget, tempus sed,
pede.

Ut odio mauris,
tincidunt ac,
molestie eu,
sagittis at,
wisi.
Ut
porttitor, est
eget accumsan
semper, neque
turpis dictum
quam,

adressFromContacts

Ut odio mauris,
tincidunt ac,
molestie eu,
sagittis at,
wisi.
Ut
porttitor, est
eget accumsan
semper, neque
turpis dictum
quam,

sendMail

fermentum lacus.
Vestibulum quam.
Duis porta,
nibh
sed
congue
tincidunt,
risus
felis
porttitor orci,
vitae pharetra erat arcu eu
dolor.
Sed lacus nulla, auctor eget,
interdum eget, tempus sed,
pede.

saveToSentFolder

Lorem ipsum dolor
sit amet, consectetuer
adipiscing elit.
Praesent
lacinia
tellus ac
orci
laoreet
tincidunt.
Phasellus dictum

Fördelar med modularisering

- Många små enheter är enklare än få större att överblicka, navigera & återanvända
- En design i flera moduler möjliggör parallell utveckling
- En moduls funktioner och data kan kapslas in
  - Förenklar återanvändning
  - Skyddar mot propagerande förändringar
  - Förbättrad underhållsbarhet

Modulariseringsstrategier

- Ett programs uppdelning i moduler kan drivas av flera olika faktorer, ex:
- Relaterade funktioner/åtaganden

  Funktioner som rör X för sig, funktioner som rör Y för sig, …
- Implementationsdetaljer

  Allt som rör nätverkskoppling ligger i en delad modul, …
- Process-pragmatika

  Allt som kräver att Kim är inblandad samlar vi i en modul, …
- Kopplingar mellan data

  Alla funktioner som bearbetar persondata i en modul, …

Modularisering är inte nominell

- Ej nominell – det blir inte en modul bara för att man säger att det är det

  Två moduler med starka interberoenden är effektivt en modul

  En modul för ”resten av funktionerna” blir inte en modul

- Coupling och cohesion hjälper till att skapa fungerande moduler

  Coupling: beroenden / koppling

  Cohesion: sammanhang

(såna här kvalitetsaspekter kan du ta fram kvantifierade mått på mha verktyg)

Vad är bra design?

- **Låg coupling**

  Interaktionen mellan moduler är så liten som möjligt

- **Höggradig inkapsling**

  En modul kan använda en annan modul, men har inte direkt åtkomst till dess interna data

- **Hög cohesion**

  Innehållet i varje modul bildar en ”logiskt vettig” enhet med hög conceptuell integritet

- Inte alltid möjligt till följd av pragmatiska skäl:

  Begränsade resurser: kompetenser hos utvecklarna, etc.

  Optimering kommer ofta på kant med i övrigt god design

Moduler i C

- Saknar motsvarande språkkonstruktion

  dvs. det finns inget nyckelord ”`module`”

- En modul är i regel en .c-fil och en (eller flera) .h-fil(er)

  .h-fil: modulens (publika) gränssnitt och definitioner

  .c-fil: implementationen (själva koden)

list.c

```c
struct list {
  struct link *first, *last;
};

struct link {
  int value;
  struct link *next;
};

typedef struct list *List;

int length(List);
int empty(List);
struct link* mkLink(…);

void append(...) {
  ...
}

int length(...) {
  ...
}

int empty(...) {
  ...
}

List mkList() {
  ...
}

struct link* mkLink(...) {
...
}
```

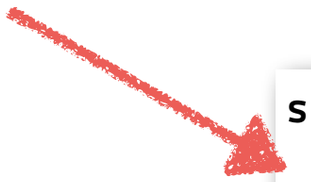
Deklarationer oftast högst upp i filen (varför?)

Funktionsprototyper för ”hjälp-funktioner”

”Själva koden”

Innan modularisering

(Vi återkommer till var dessa skall ligga senare!)



list.h

```
struct list {
  struct link *first, *last;
};

struct link {
  int value;
  struct link *next;
};

typedef struct list *List;

void append(List, int);
int length(List);
int empty(List);
List mkList();
struct link* mkLink(...);
```

Deklarationer & funktionsprototyper

list.c

```
#include "list.h"

void append(...) {
  ...
}

int length(...) {
  ...
}

int empty(...) {
  ...
}

List mkList() {
  ...
}

struct link* mkLink(...) {
```

”Själva koden”

(Vi återkommer till var dessa skall ligga senare!)

list.h

```
struct list {
  struct link *first, *last;
};

struct link {
  int value;
  struct link *next;
};

typedef struct list *List;

void append(List, int);
int length(List);
int empty(List);
List mkList();
struct link* mkLink(...);
```

list.c

```
#include "list.h"

void append(...) {
  ...
}

int length(...) {
  ...
}

int empty(...) {
  ...
}

List mkList() {
  ...
}

struct link* mkLink(...) {
```

#include-direktiv kopierar in innehållet i inkluderade filer vid kompilering

list.h

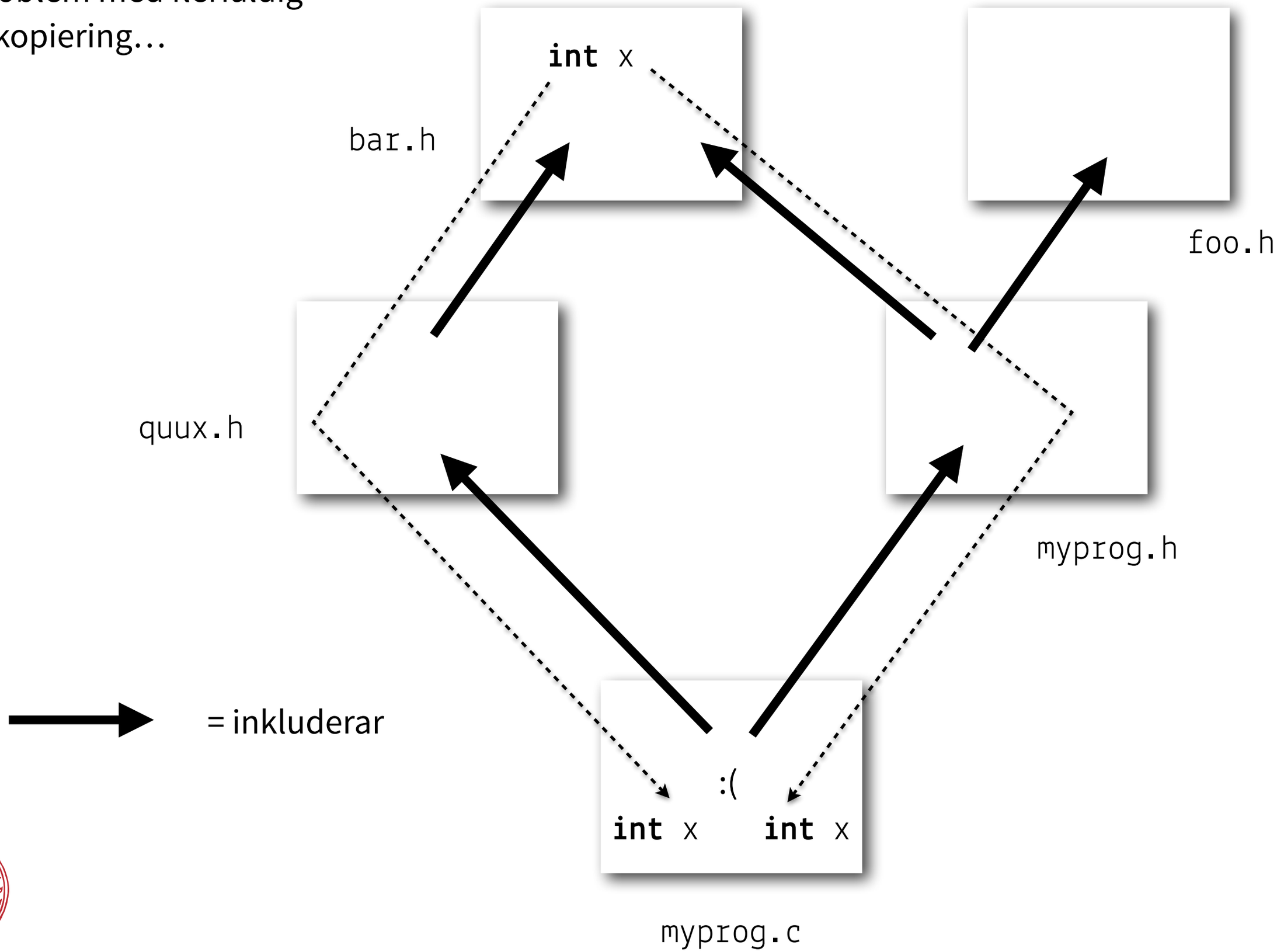
```
struct list {
  struct link *first, *last;
};

struct link {
  int value;
  struct link *next;
};

typedef struct list *List;

void append(List, int);
int length(List);
int empty(List);
List mkList();
struct link* mkLink(...);
```

list.c

```
struct list {
  struct link *first, *last;
};

struct link {
  int value;
  struct link *next;
};

typedef struct list *List;

void append(List, int);
int length(List);
int empty(List);
List mkList();

struct link* mkLink(…);

void append(...) {
  ...
}

int length(...) {
  ...
}

int empty(...) {
```

#include-direktiv kopierar in innehållet i inkluderade filer vid kompilering

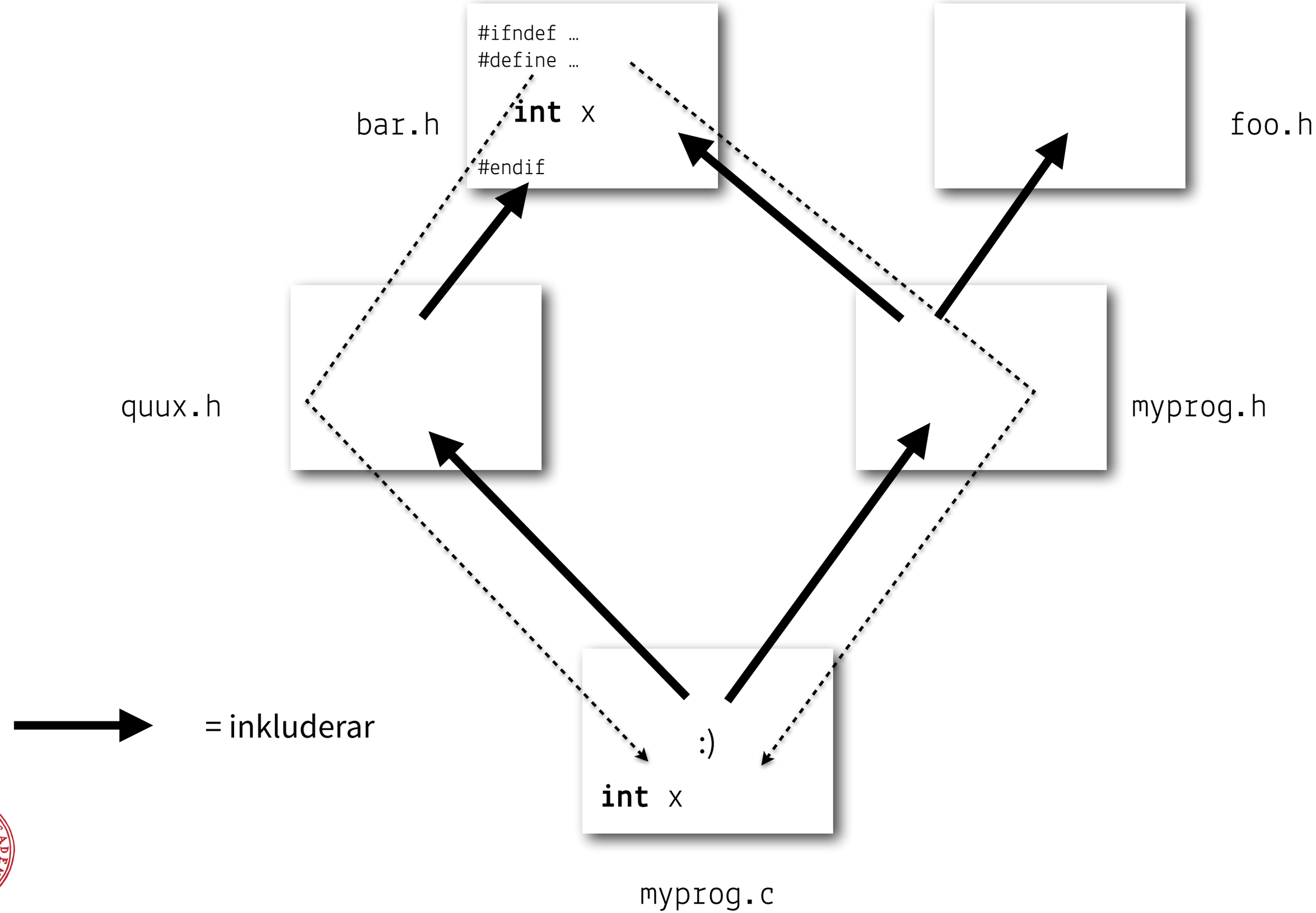
problem med flerfaldig inkopiering…



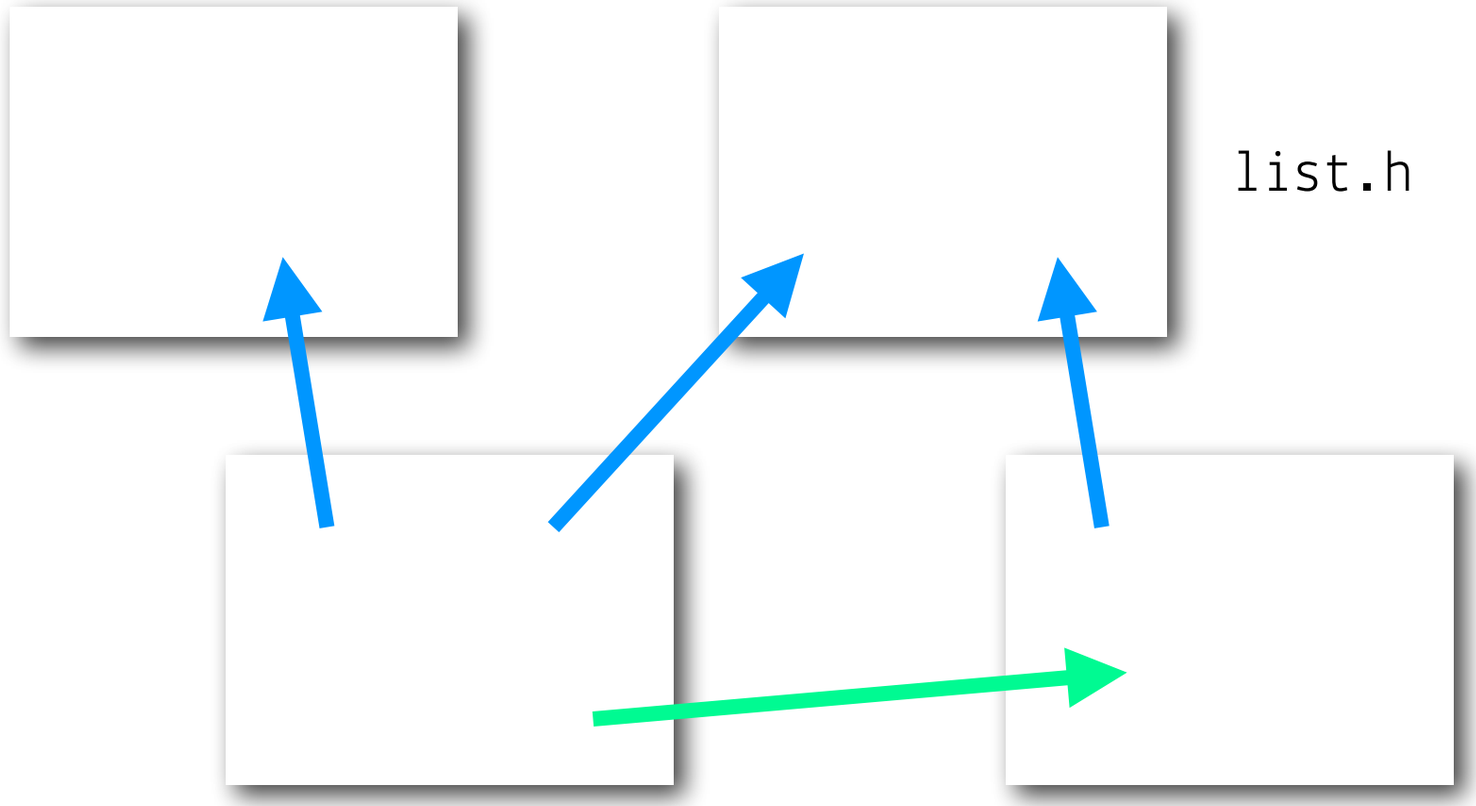
list.h

förhindrar flerfaldig inkopiering

```
#ifndef __list_h
#define __list_h

struct list {
  struct link *first, *last;
};

struct link {
  int value;
  struct link *next;
};

typedef struct list *List;

void append(List, int);
int length(List);
int empty(List);
List mkList();
struct link* mkLink(...);

#endif
```


#ifndef …
#define …
int x
#endif
bar.h
foo.h
quux.h
myprog.h
:)
int x
myprog.c
= inkluderar

myprog.h



list.h

myprog.c

list.c

= kompileringsberoende

= länkningsberoende

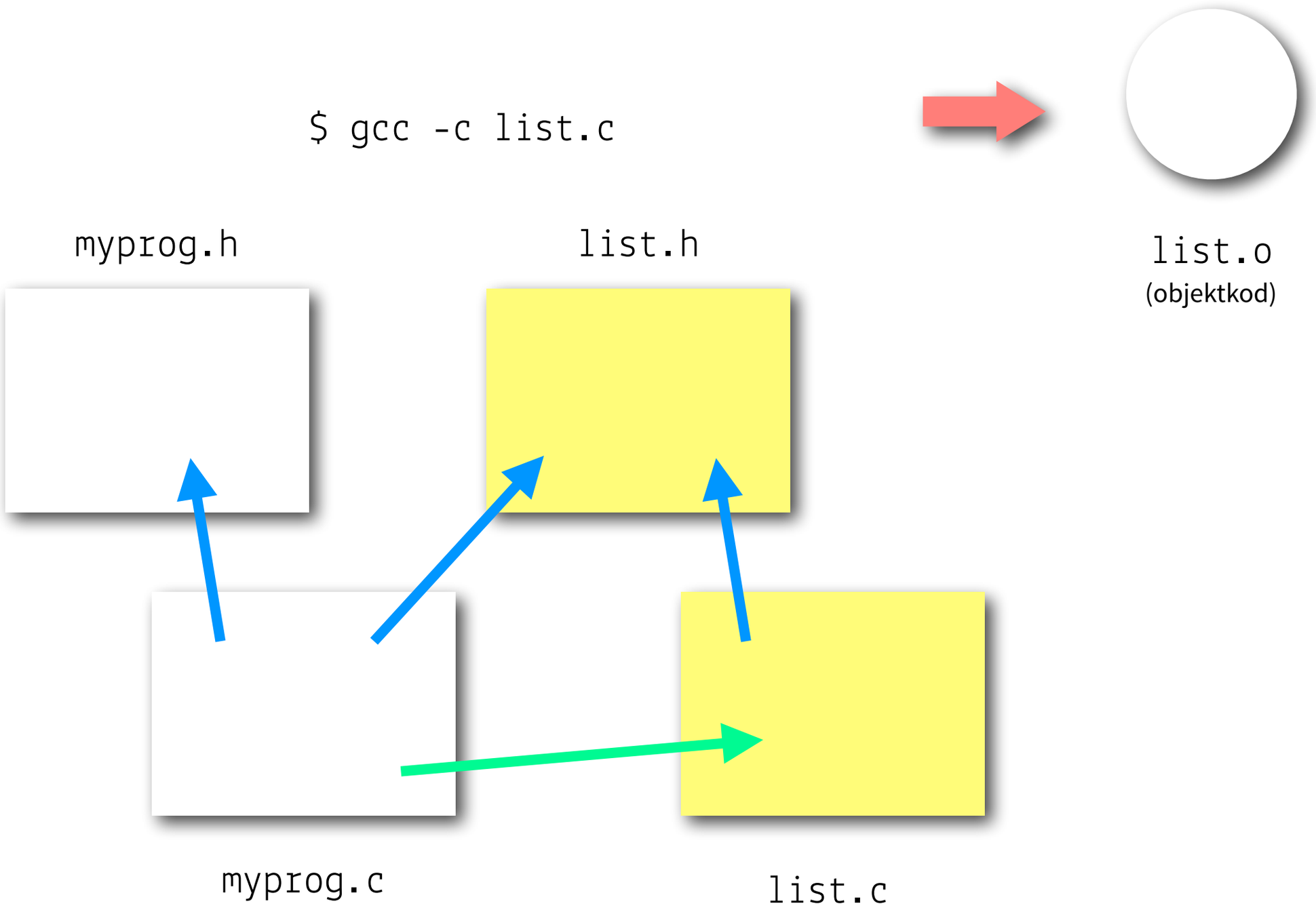
Separatkompilering och länkning

- Separatkompilering producerar ofullständig objektkod
- Möjliggör kompilering av delar av program till objektkod

  Kompilering medger statisk felkontroll

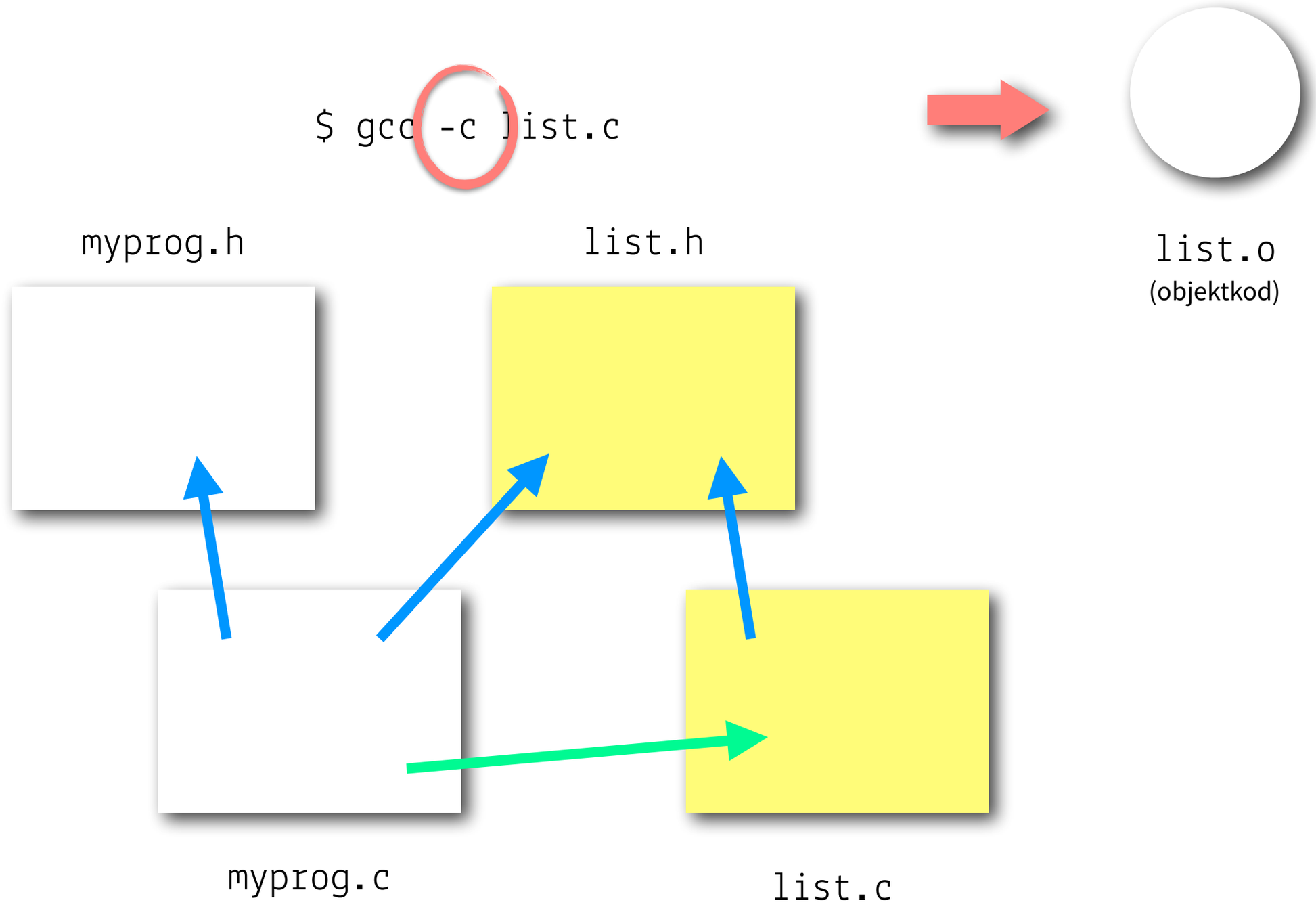
  Objektkoden kan vidare distribueras
- Alla separatkompilerade moduler länkas slutligen ihop till ett körbart program

  Länkningen löser ut beroenden mellan modulerna



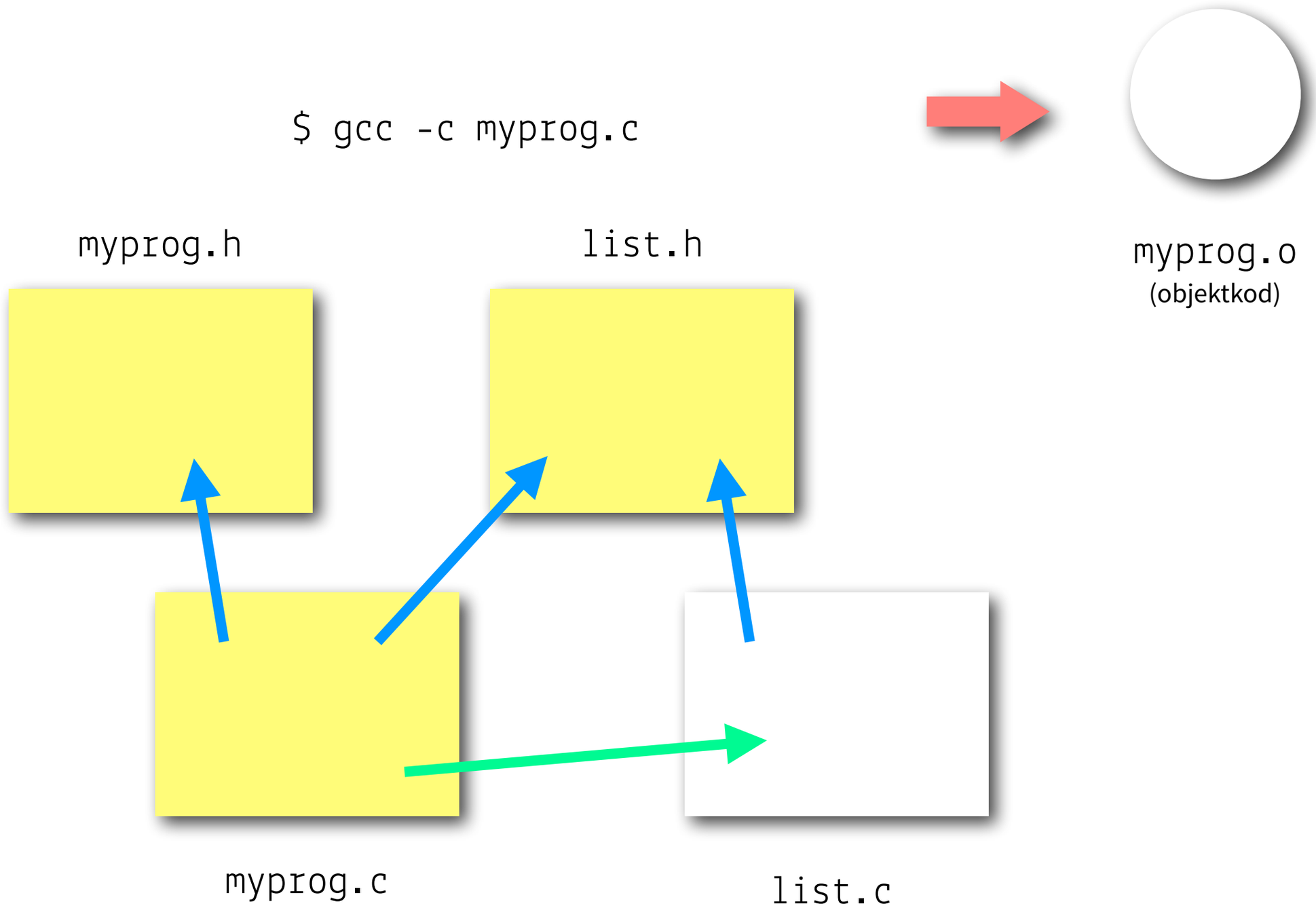
= kompileringsberoende

= länkningsberoende



= kompileringsberoende

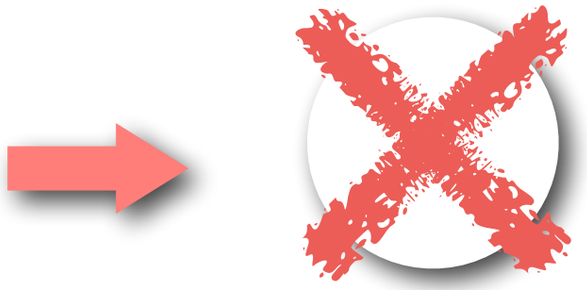
= länkningsberoende



= kompileringsberoende

= länkningsberoende

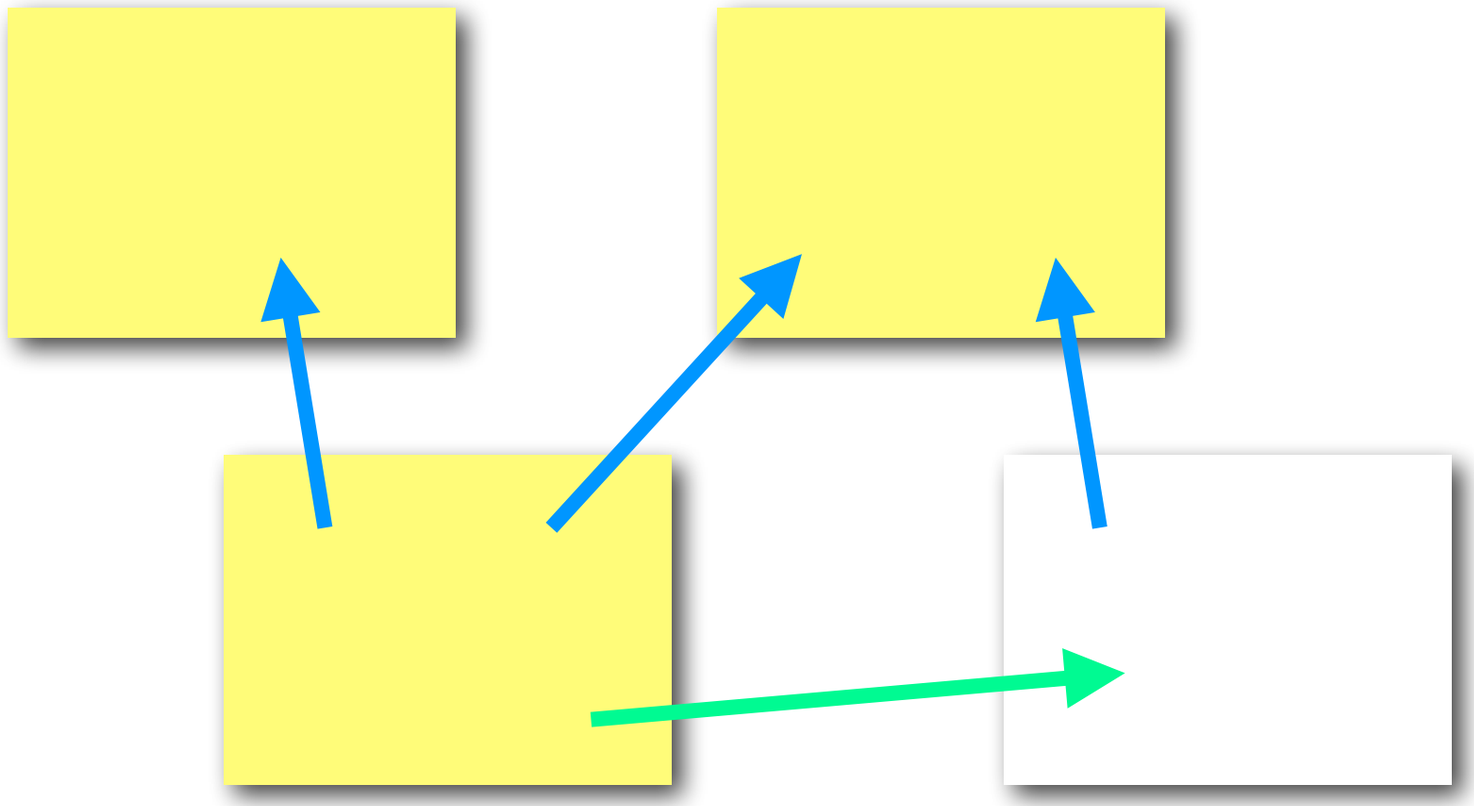
$ gcc myprog.c



a.out

myprog.h

list.h



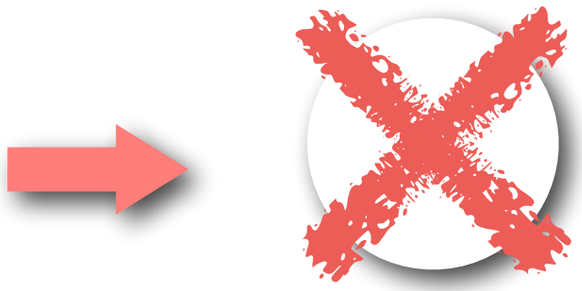
myprog.c

list.c

= kompileringsberoende

= länkningsberoende

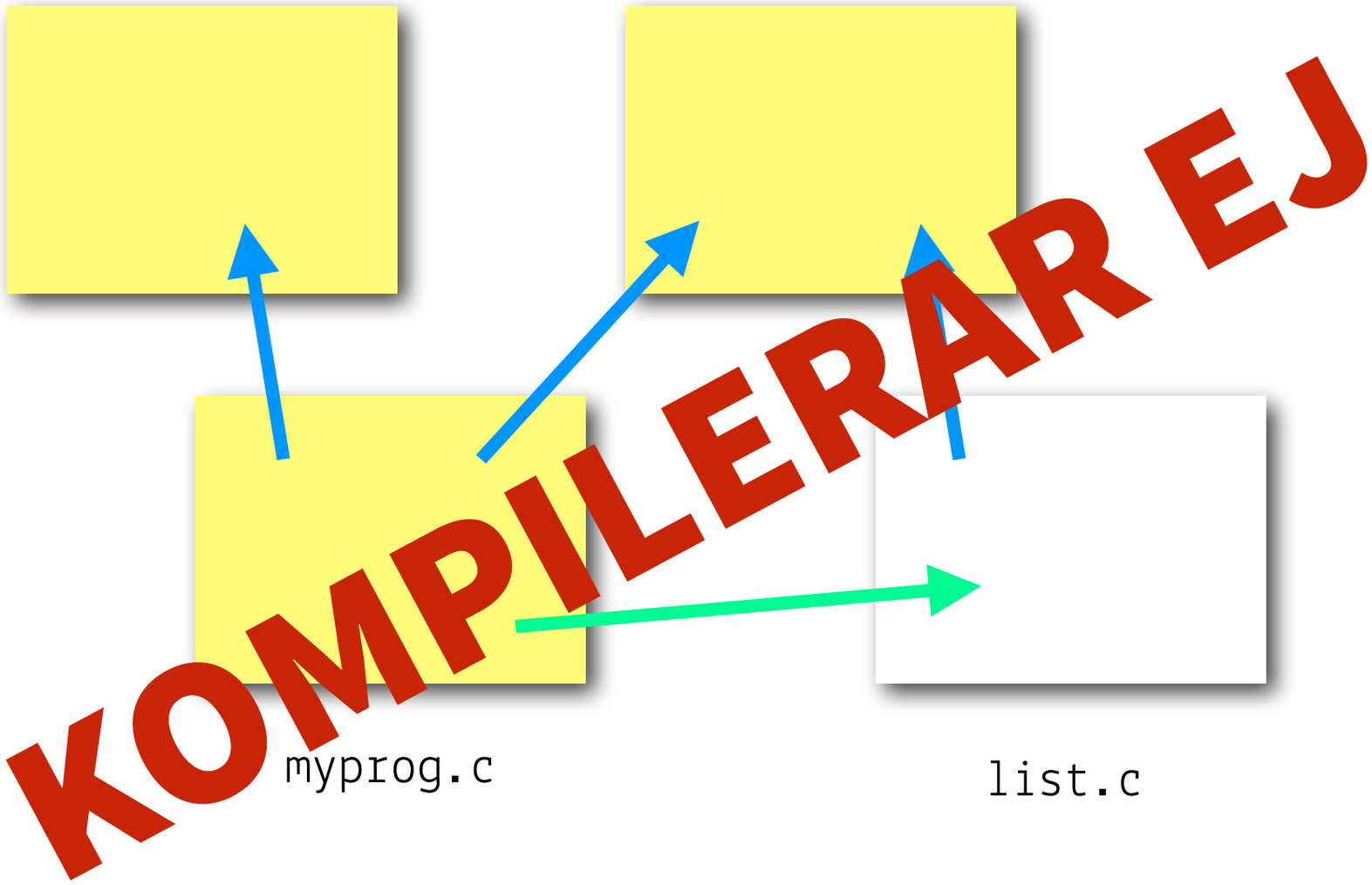
$ gcc myprog.c



a.out

myprog.h

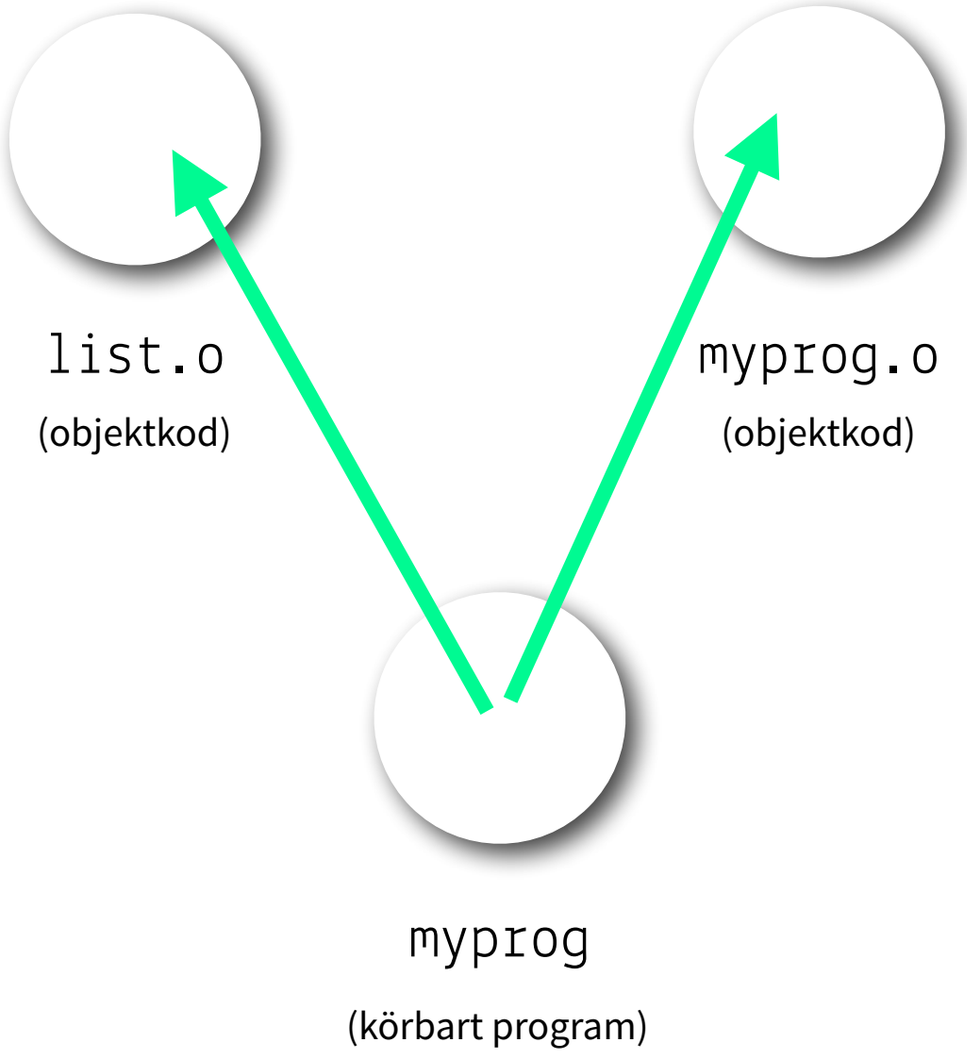
list.h



= kompileringsberoende

= länkningsberoende

```
$ gcc -o myprog list.o myprog.o
```



Coupling och cohesion

- En moduls cohesion är ett mått på utsträckningen i vilken dess åtaganden tillsammans ”ger mening” — ju högre desto bättre

  Låg cohesion betyder att en modul har väldigt många olika åtaganden

  Anti-pattern: ”god classes” (god modules)

  En modul med bara en funktion som utför en sak har maximal cohesion

- Coupling mellan moduler är ett mått på deras ömsesidiga beroende av varandra — lägre är bättre

  Hög coupling betyder att det är svårt att isolera förändringar

- Designprincip: öka cohesion och minska coupling!

De värsta sorternas coupling och cohesion

- Coincidental Cohesion

  Modulens beståndsdelar är helt orelaterade

- Content/Pathological Coupling

  När en metod använder eller förändrar data inuti en annan modul direkt, utan att gå via dess funktioner

```
commodity_t book = {};
book.cost = -12,50;
int cost_of_book = book.cost;
```

```
commodity_t book = mk_book(...);
set_cost_of_book(book, 12,50);
```

mylogging_system.c

```
void searchMessages(char* msg) { ... }

File openFile(char* fileName) { ... }

char *readFromFile(File file, int size) { ... }

void closeLogFile() { ... }

void flushLogs(char* msg) { ... }

int writeToFile(File file, char *bytes) { ... }

void logMessage(char* msg) { ... }

void deleteMessage(char* msg) { ... }

void openLogFile() { ... }

void setLogFileName(char *fileName) { ... }
```

loggning

filhantering

mylogging_system.c

```
void searchMessages(char* msg) { ... }

File openFile(char* fileName) { ... }

char *readFromFile(File file, int size) { ... }

void closeLogFile() { ... }

void flushLogs(char* msg) { ... }

int writeToFile(File file, char *bytes) { ... }

void logMessage(char* msg) { ... }

void deleteMessage(char* msg) { ... }

void openLogFile() { ... }

void setLogFileName(char *fileName) { ... }
```

loggning

filhantering

logging.c | h

```c
void searchMessages(char *msg) { ... }
void closeLogFile() { ... }
void flushLogs(char *msg) { ... }
void logMessage(char *msg) { ... }
void deleteMessage(char *msg) { ... }
void openLogFile() { ... }
void setLogFileName(char *fileName) { ... }
```

```c
File openFile(char *fileName) { ... }
char *readFromFile(File file, int size) { ... }
int writeToFile(File file, char *bytes) { ... }
```

file_handling.c | h

Informationsgömning

- En moduls implementationsdetaljer skall inte vara möjliga att observera utifrån

- Designprincip:

  Göm föränderliga detaljer bakom ett stabilt gränssnitt

- Inkapsling är en term som ofta används synonymt med informationsgömning

  Man kan se inkapsling som en teknik, informationsgömning som en princip

Sammanfattning

- Att ett program är korrekt och effektivt är bara två egenskaper av många som ett bra program skall ha
- Använd alltid lämpliga abstraktioner för att göra program överskådliga och enklare att ändra (kontrollabstraktion och dataabstraktion, t.ex.)
- Designprincipen modularisering är ett viktigt verktyg för att bryta ned ett problem i (allt) mindre beståndsdelar som blir enklare att lösa

  Dela alltid upp era program i moduler
- Designprincipen informationsgömning är viktig för att skydda abstraktioner

  Ge en modul ett stabilt gränssnitt (i en .h-fil i C)
- Inkapsling är en viktig teknik för informationsgömning

  Exponera aldrig interna funktioner eller struktdefinitioner i gränssnittet (.h-filen)