

# Föreläsning 4, del 1

---

Tobias Wrigstad

*Poster, unioner  
och typedef*



# 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" (Date 1987 3 4)
```

# Sammansatta datatyper i C

värdesemantik!

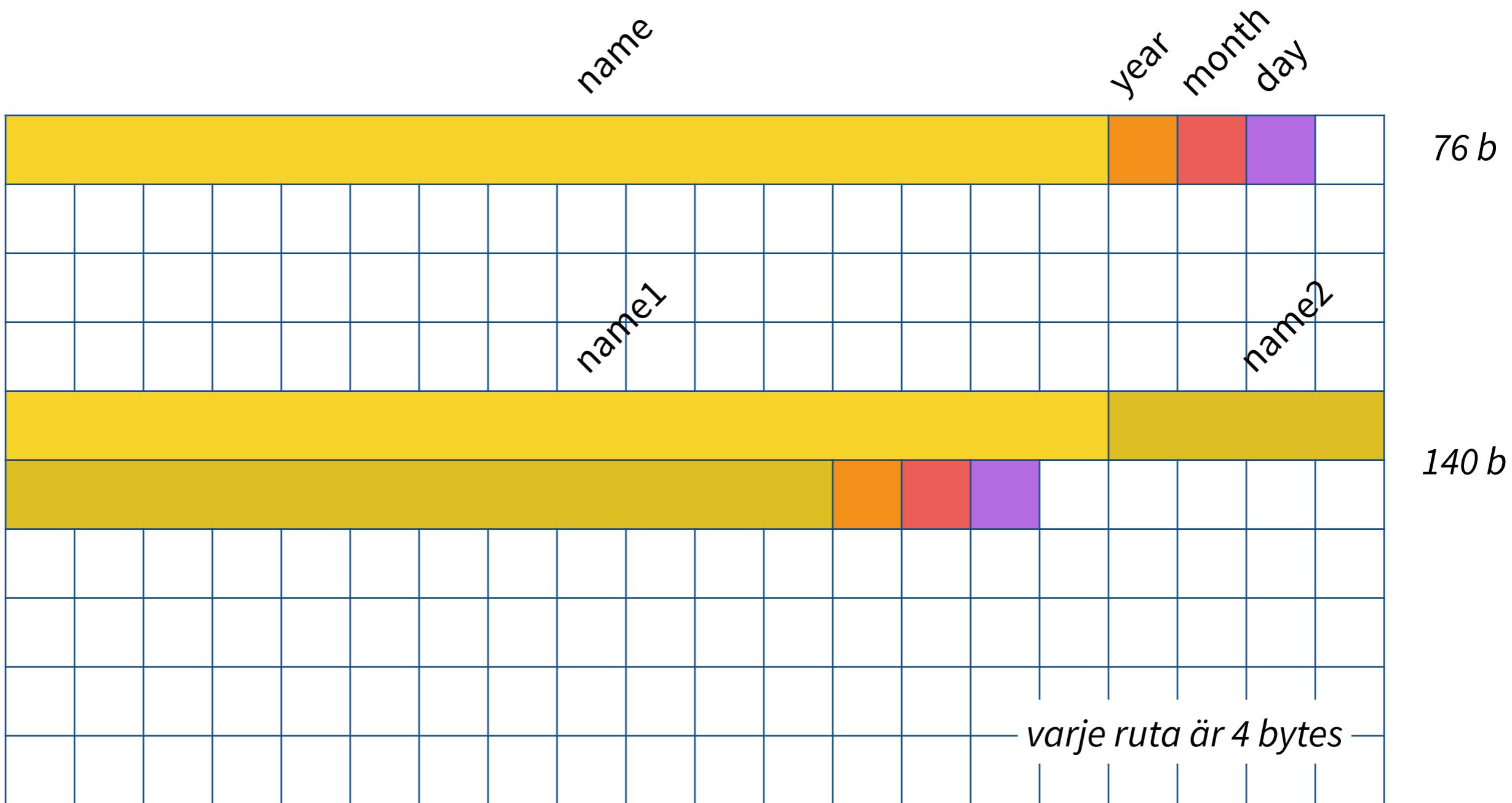
- Ingen pattern matching
- Inget enkelt sätt att konstruera eller dekonstruera
- Definitioner för att skapa ett sammanhängande datablock med olika *namngivna fält* som innehåller *värden av fix storlek*.

Ingen enkel motsvarighet till Anniversary

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

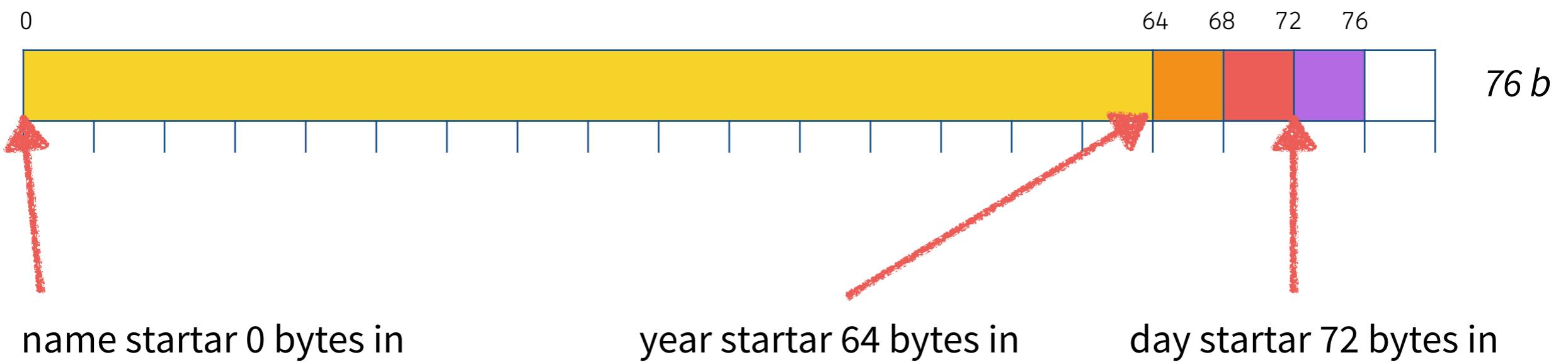
```
struct wedding
{
    char spouse_name1[64];
    char spouse_name2[64];
    int year;
    int month;
    int day;
};
```

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



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

---



# Unions – inte riktigt algebraiska datatyper...

---

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

Storleken (**sizeof**) på en union är  $\max(\text{sizeof}(T_1), \text{sizeof}(T_2))$

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 är  $\max(\text{sizeof}(T_1), \text{sizeof}(T_2))$

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.w.name = ...
    }
    else
    {
        a.b.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; // uninitialized 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

---

```
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!

```
#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!*

```
#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;
}
```



# Typedef

```
#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;
}
```

*Vi kommer att använda  
typedef från och med nu*

```
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 4, del 2

---

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. fackterminer

# 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



(nästan alltid irrelevant)

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

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

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

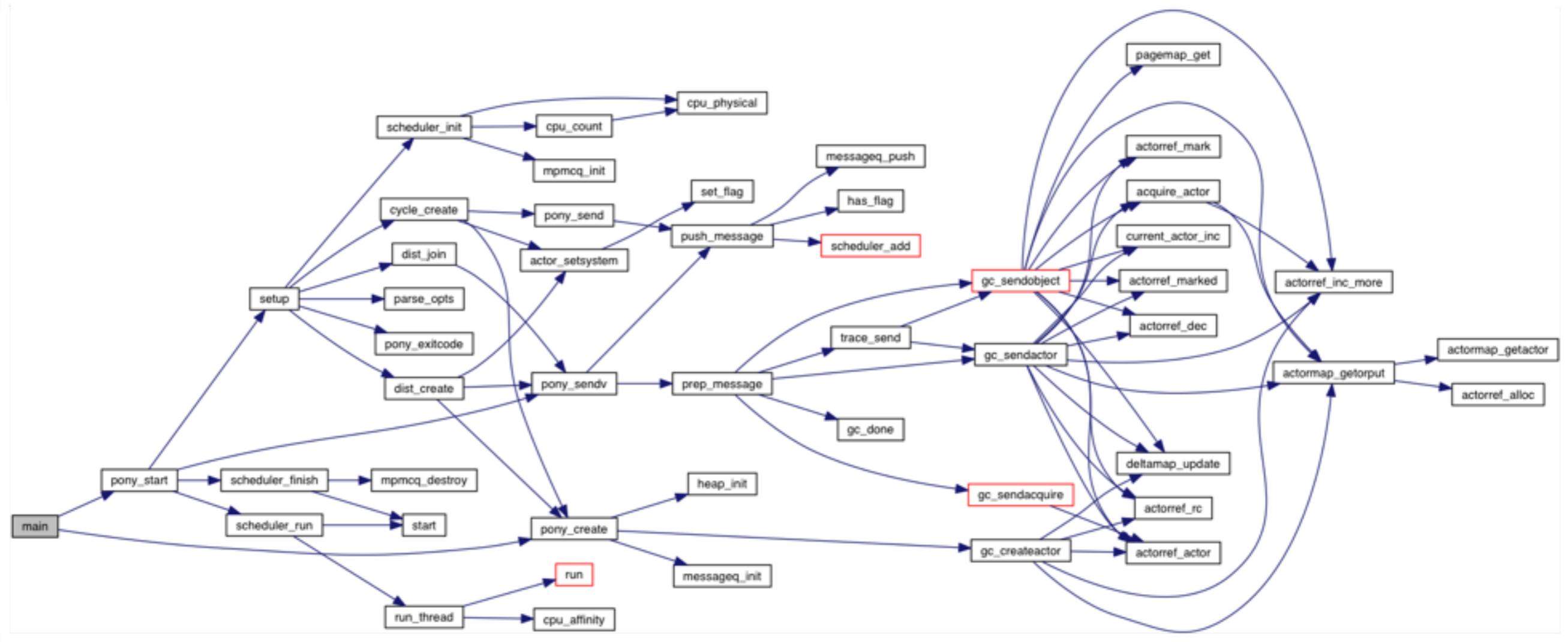
fermentum lacus.  
Vestibulum quam.  
Duis porta,  
nibh  
sed  
congue  
tincidunt,  
risus  
felis  
porttitor orci,  
vitae pharetra erat arcu eu  
dolor.  
Sed lacus nulla, auctor eget,  
interdum eget, tempus sed,  
pede.

## saveToSentFolder

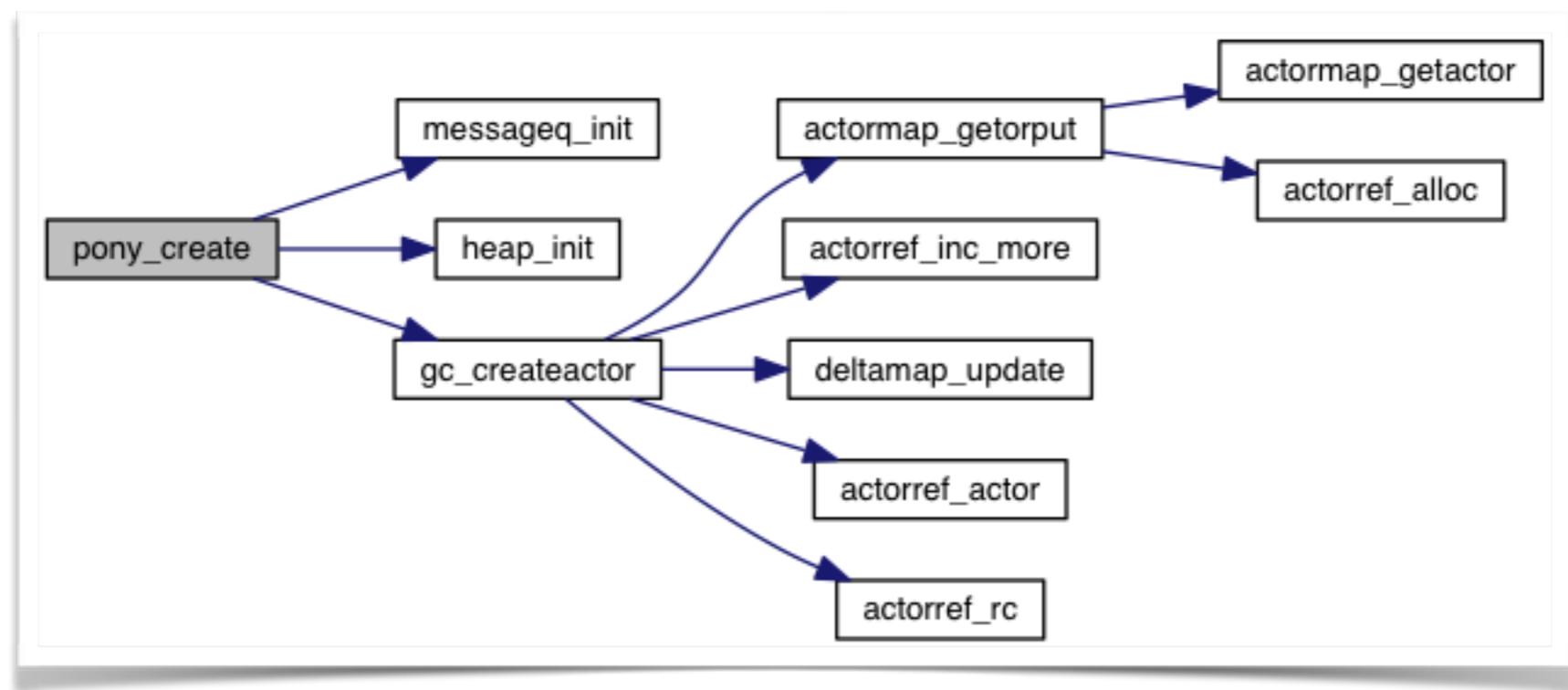
Ut odio mauris  
tincidunt ac  
molestie eu  
sagittis at  
wisi.  
Ut  
porttitor, est  
eget accumsan  
semper, neque  
turpis dictum  
quam,  
Phasellus dictum



# Anropsgraf

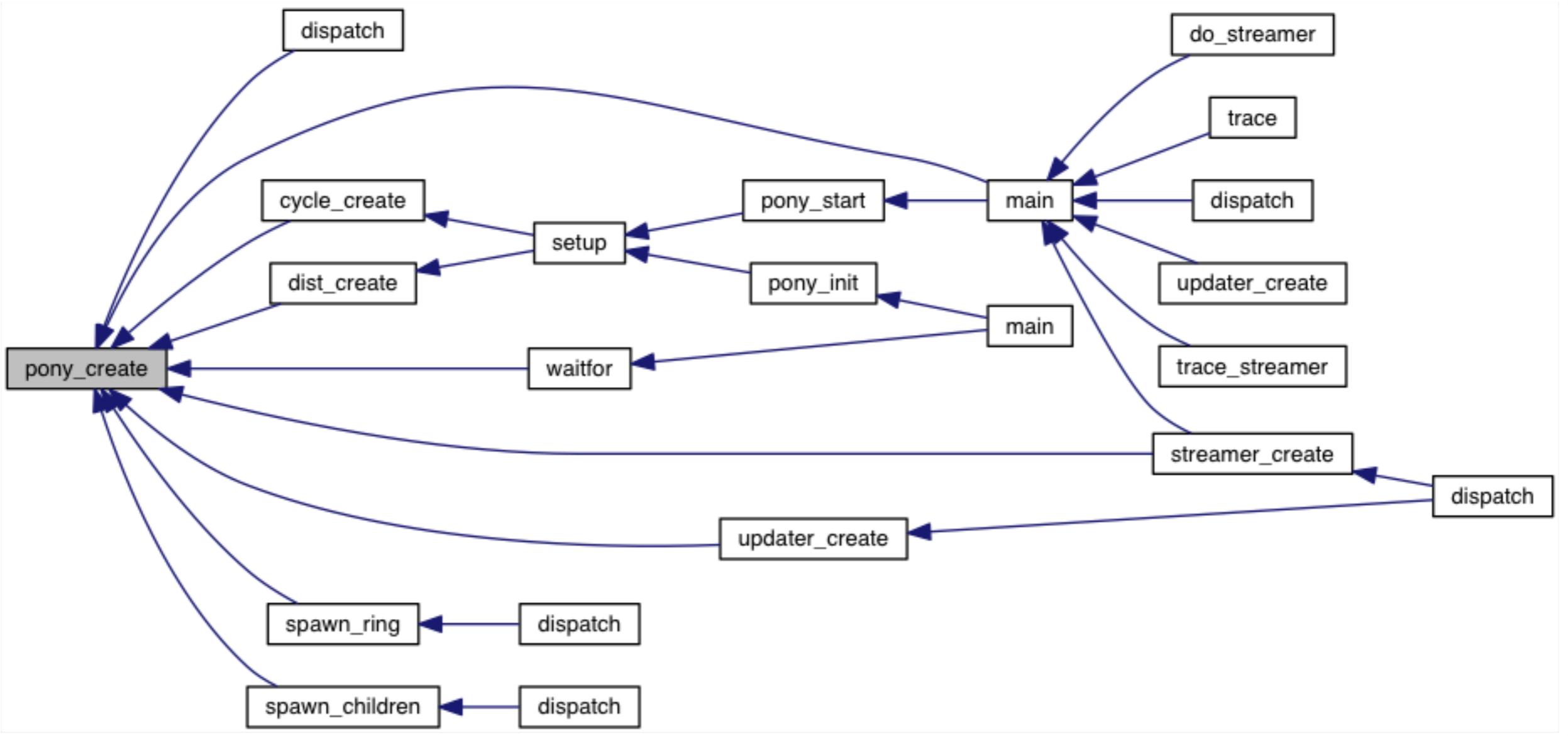


Doxygen ("vårt" dokumentationsverktyg för C) kan generera statiska anropsgrafer



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”

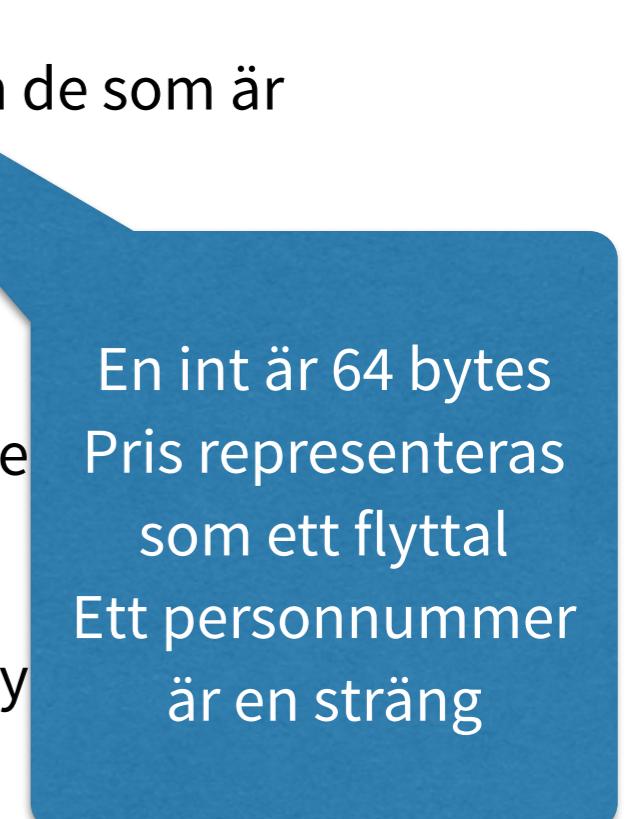
Programmen bör inte förutsätta att datat är implementerat på ett särskilt vis

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 oberoende av program som använder datat

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

Kapsla in implementationsdetaljer och undvik externa beroenden



En int är 64 bytes  
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 `commodity_t *book = ...;` t särskilt vis

```
commodity_t *book = ...;  
book->cost = 12.50;
```

Programmen bör i `int cost_of_book = book->cost;` in 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 i `commodity_t *book = ...;` t särskilt vis

Programmen bör i `int cost_of_book = book->cost;` in de som är  
nödvändiga för att utföra den aktuella uppgiften

- Abstrakt `int cost_in_eurocent(commodity_t *com);`
  - Konkret `void set_cost_in_eurocent(commodity_t *com, int cost);`
- program som använder datat

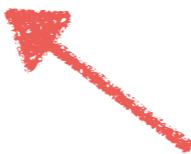
Använd accessorer och mutatorer för manipulera datat som blir bryggan mellan  
det abstrakta och konkreta datat

Kapsla in implementationsdetaljer och undvik externa beroenden

# 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 programspecifik (jmf. t.ex. stdlib i C; eller en lista)

ponyrt

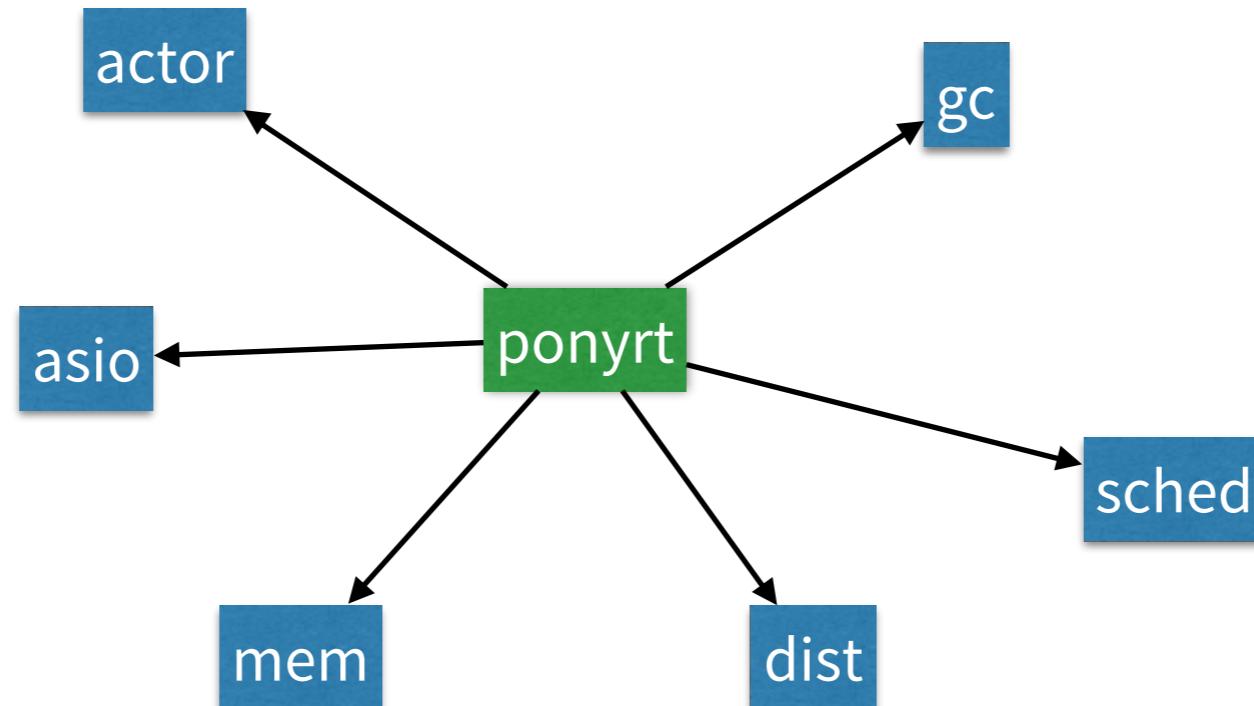
# Modularisering

---

- 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 programspecifik (jmf. t.ex. stdlib i C; eller en lista)



# Fördelar med modularisering

---

- Många små enheter är enklare än 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

**L**orem ipsum dolor  
sit amet, consectetur  
adipiscing elit.  
**P**raesent  
lacinia  
tellus ac  
orci  
laoreet  
tincidunt.  
**P**hasellus dictum

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

**L**orem ipsum dolor  
sit amet, consecetetuer  
adipiscing elit.  
**P**raesent  
lacinia  
tellus ac  
orci  
laoreet  
tincidunt.  
**P**hasellus 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 vi måste ha Åsa till att skriva 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

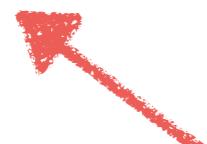
Två moduler med starka interberoenden är effektivt en modul

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)

# Ett lackmus-test för 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

```
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"



(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(...);
```



Deklarationer &  
funktionsprototyper



”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(...) {
    ...
}

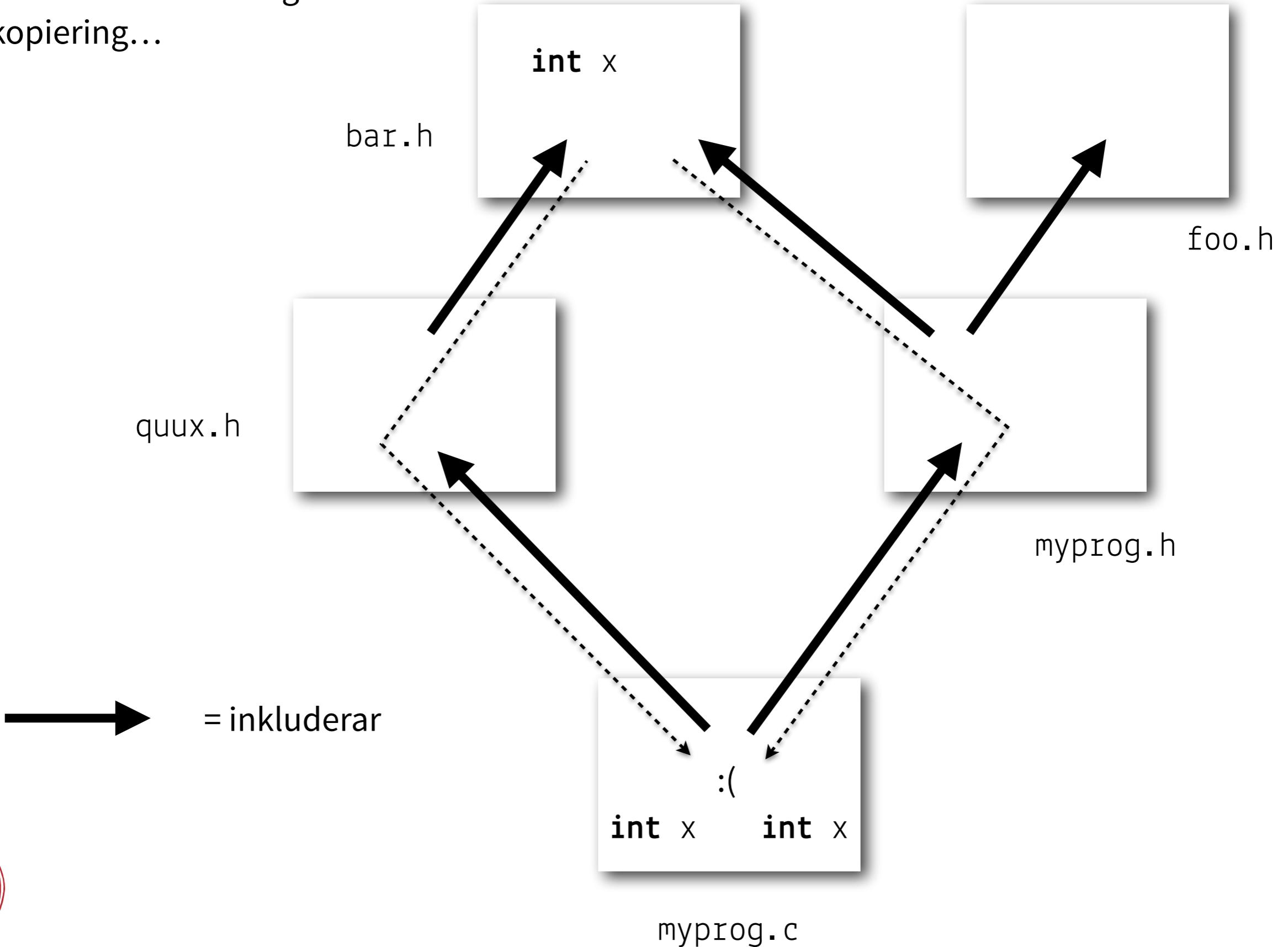
int length(...) {
    ...
}

int empty(...) {
```

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



problem med flerfaldig  
inkopiering...



förhindrar  
flerfaldig  
inkopiering

list.h

```
#ifndef __list_h
#define __list_h

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

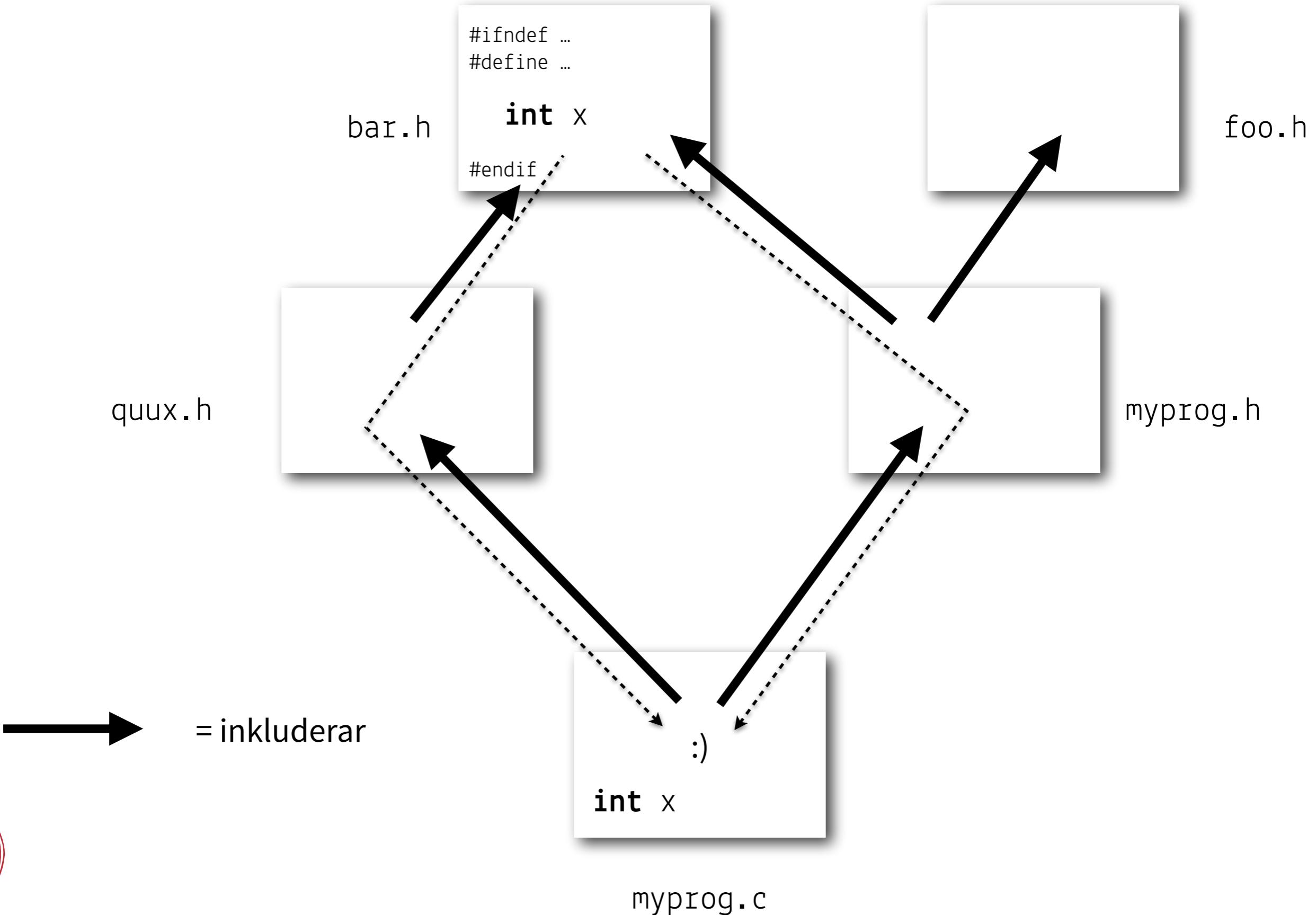
struct link {
    int value;
    struct link *next;
};

typedef struct list *List;

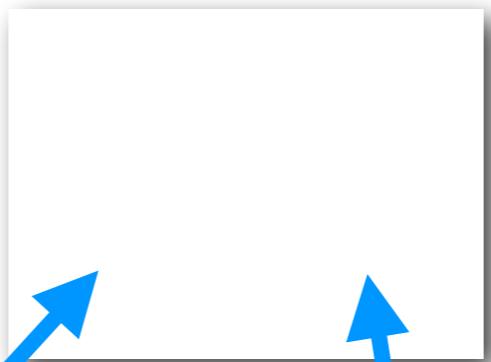
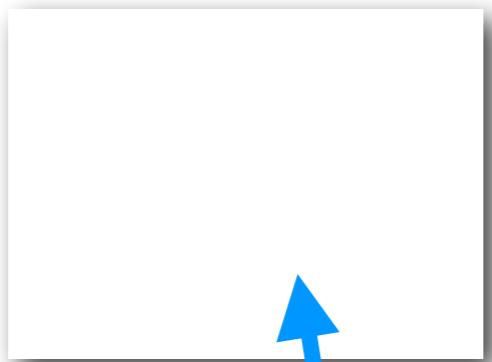
void append(List, int);
int length(List);
int empty(List);
List mList();
struct link* mkLink(...);

#endif
```





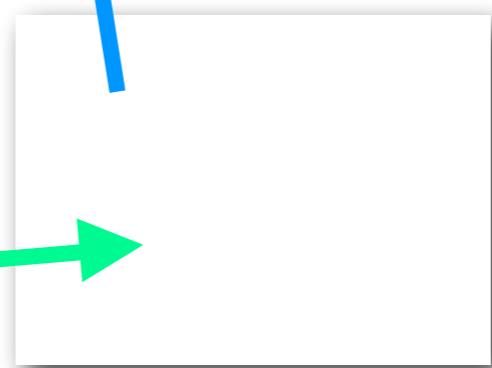
myprog.h



list.h



myprog.c



list.c

→ = kompileringsberoende

→ = länkningsberoende



# Separatkompilering och länkning

---

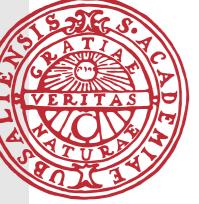
- Separatkompilering producerar ofullständig objektkod
- Möjliggör kompilering av delar av program till objektkod

Kompilering medger statisk felkontroll

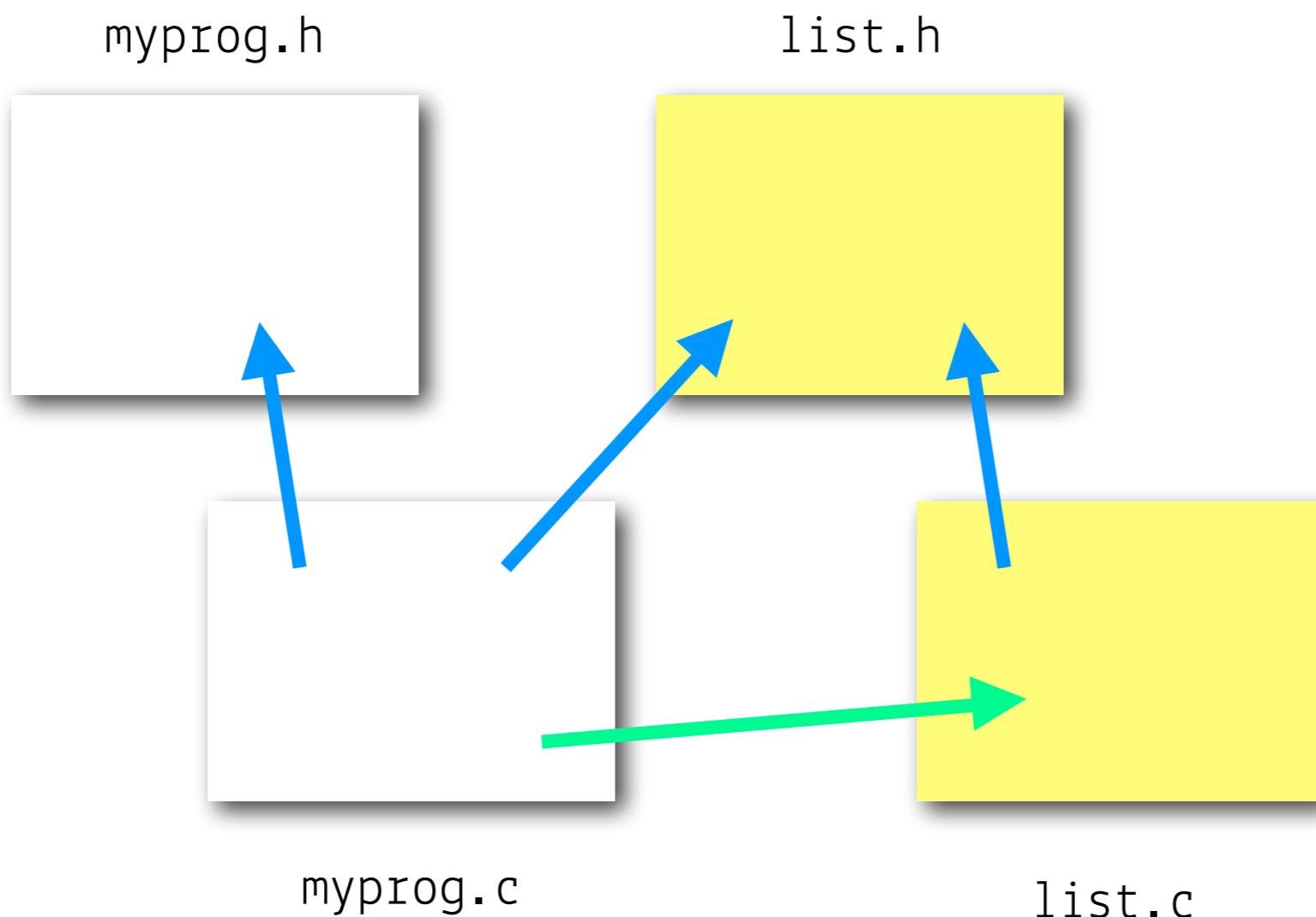
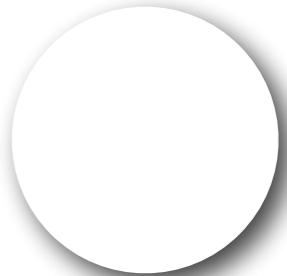
Objektkoden kan vidare distribueras

- Alla separatkompilerade moduler länkas slutligen ihop till ett körbart program

Länkningen löser ut beroenden mellan modulerna

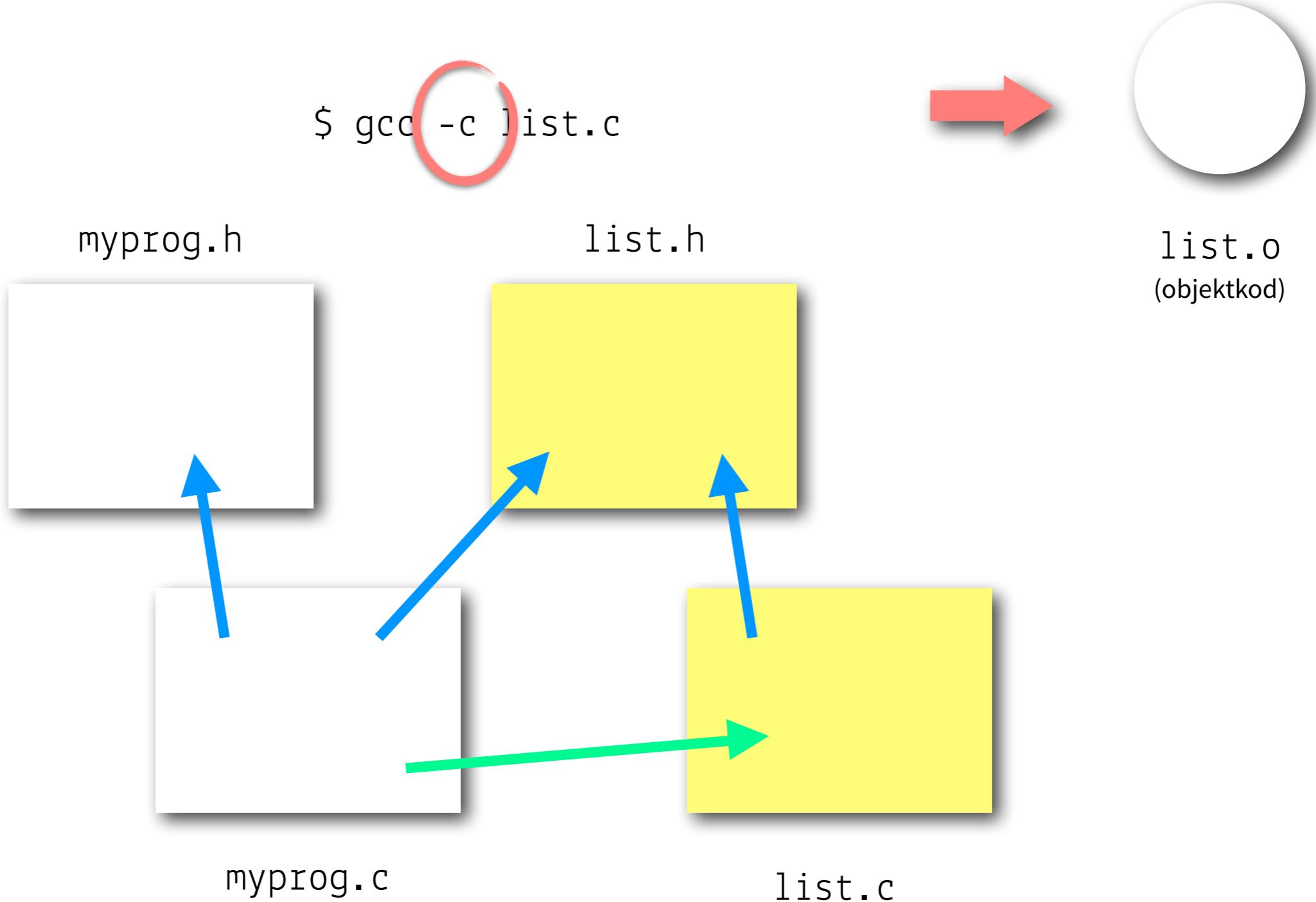


```
$ gcc -c list.c
```



→ = kompileringsberoende

→ = länkningsberoende

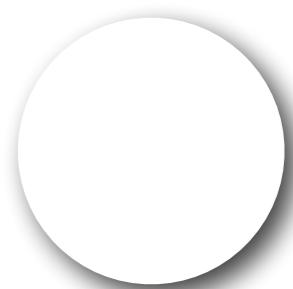


→ = kompileringsberoende

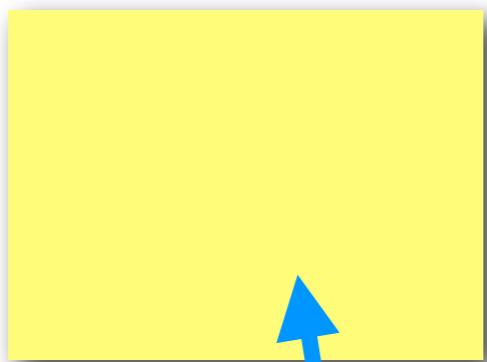
→ = länkningsberoende



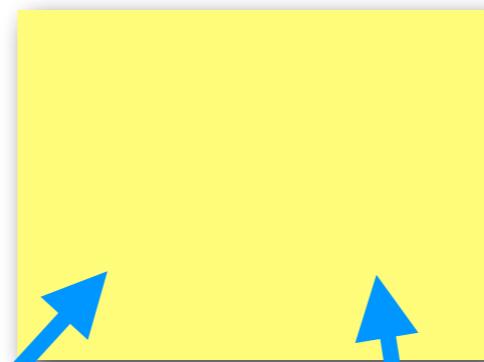
```
$ gcc -c myprog.c
```



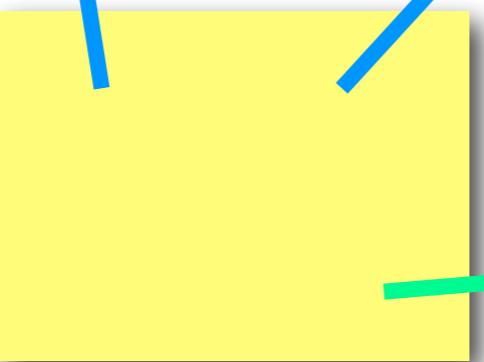
myprog.h



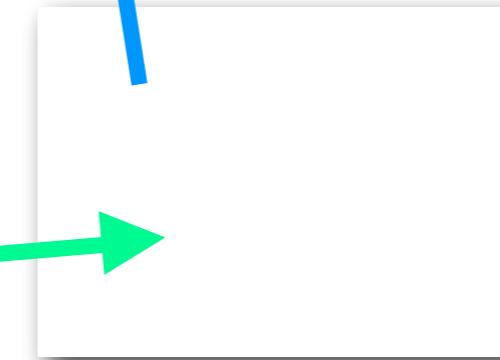
list.h



myprog.o  
(objektkod)



myprog.c

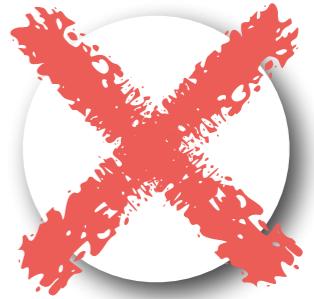


list.c

→ = kompileringsberoende

→ = länkningsberoende





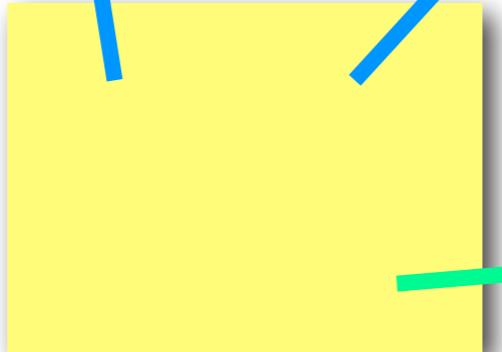
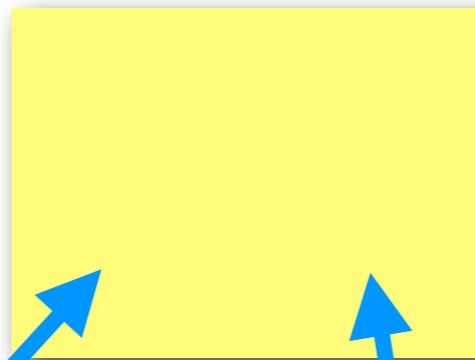
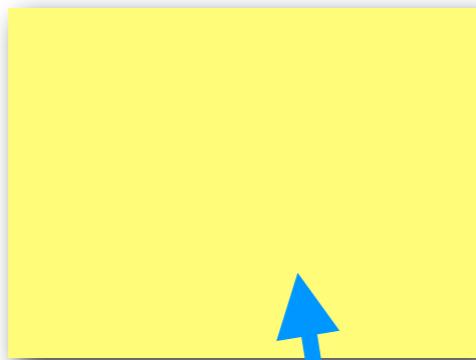
\$ gcc myprog.c



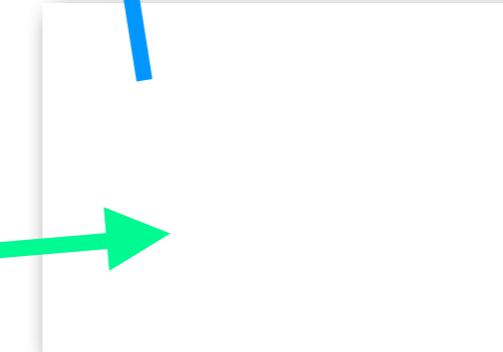
a.out

myprog.h

list.h



myprog.c



list.c

→ = kompileringsberoende

→ = länkningsberoende





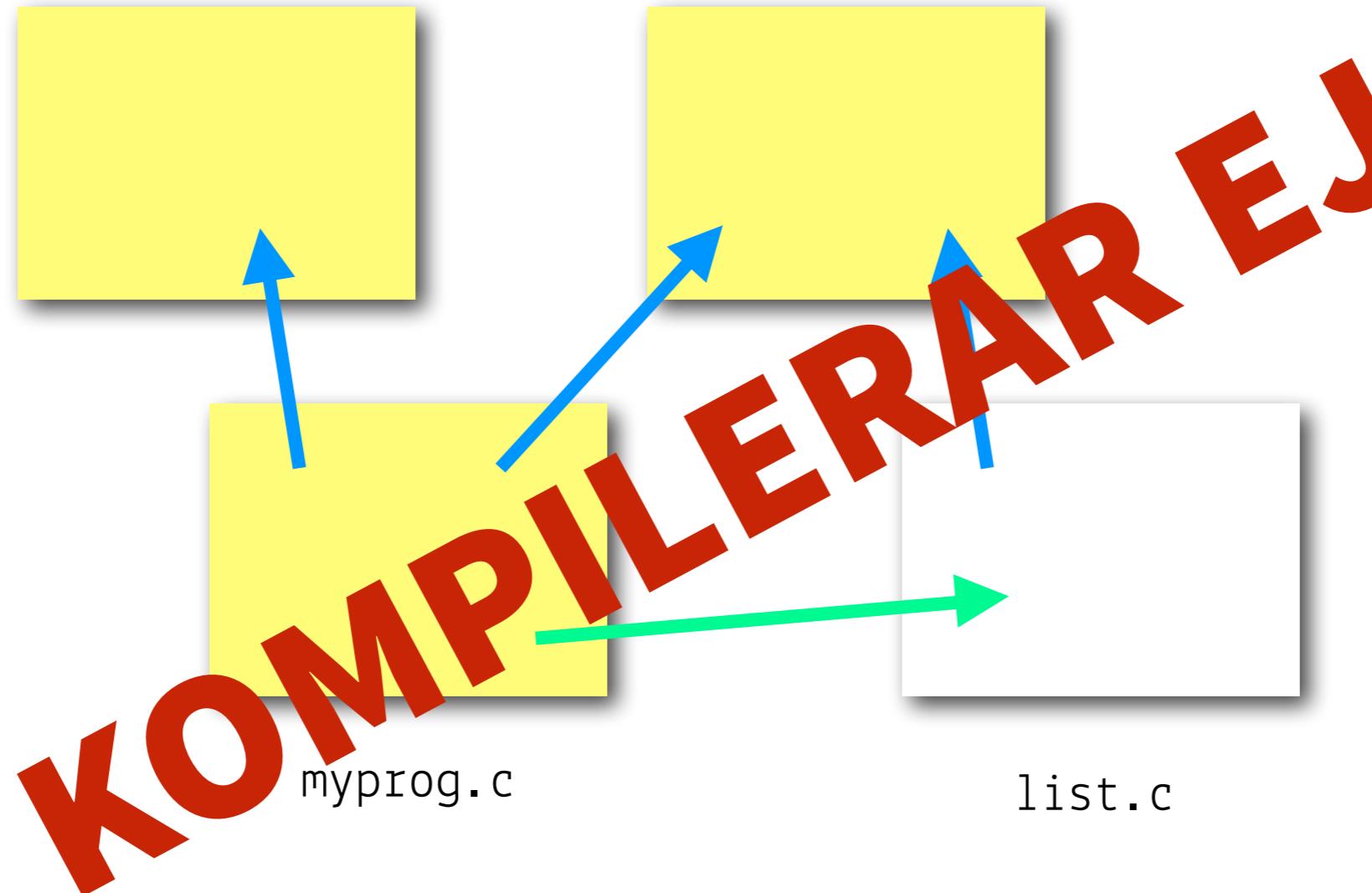
\$ gcc myprog.c



myprog.h

list.h

a.out

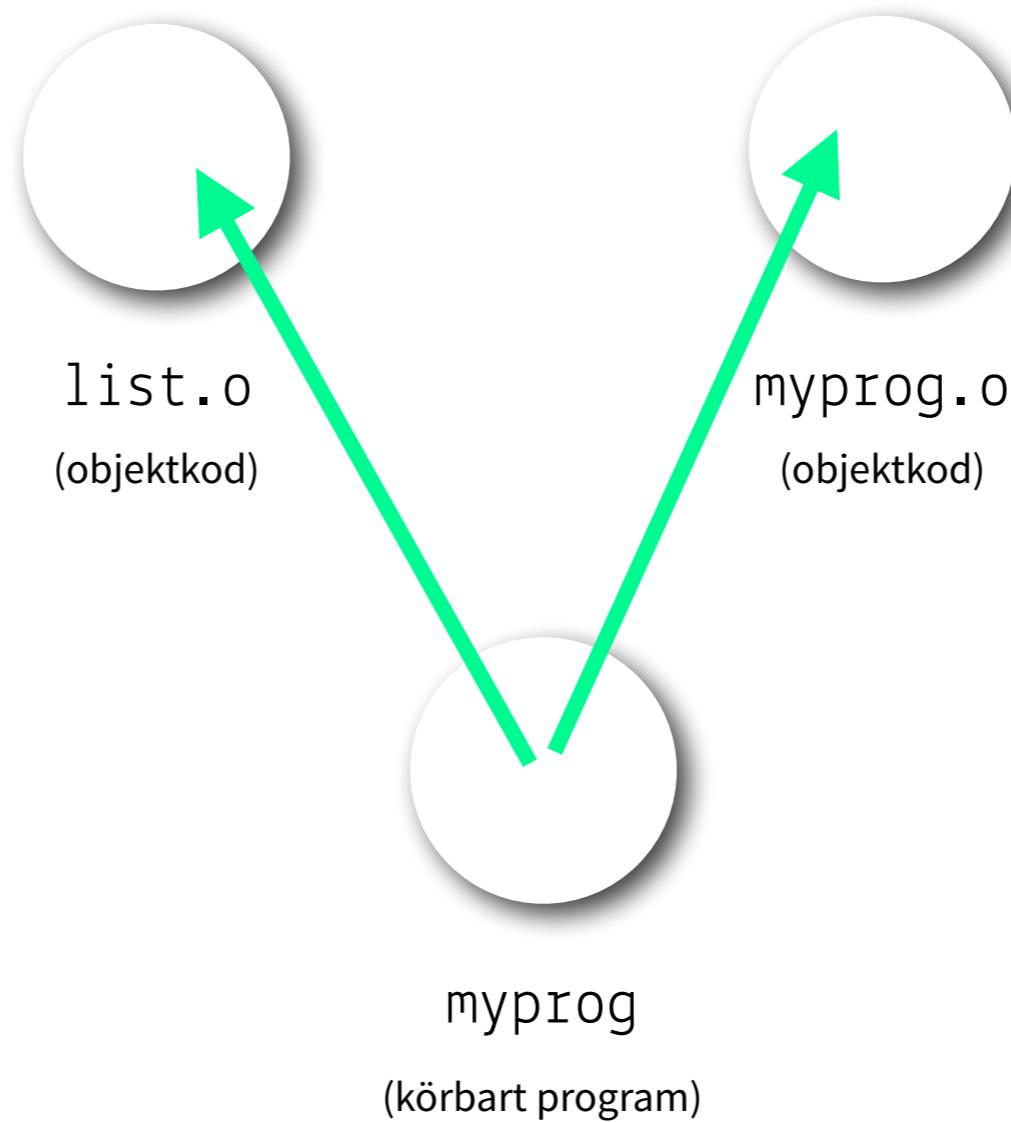


→ = kompileringsberoende

→ = 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;
```

## 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) { ... }
```

*loggnig*

*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) { ... }
```

logging

filhantering



## logging.c | h

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

```
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

# Exempel på informationsgömning: en lista

---

- Något förenklat kan man säga att alla listor tillhandahåller samma tjänster, d.v.s. man kan stoppa in element i listan, ta bort, etc.

Samma tjänster = samma (stabila) gränssnitt

- Hur listan är implementerad är av oerhörd vikt för icke-funktionella aspekter av ett program, t.ex.

En lista som är implementerad med en array är mer effektiv att iterera över än en länkad lista pga god lokalitet (map prestanda är den nästan alltid bättre)

 (sällan viktigt!)

En länkad lista kan vara snabbare att göra insättningar i början på listan än en array eftersom den senare måste ”knuffa alla element ett steg”

- Att byta från en typ av lista till en annan bör inte kräva förändringar mer än på den rad där en lista skapas



Det här är det som är viktigt!

```
// list.h  
typedef struct list *List;  
  
List mkList();  
void append(List, int);  
void prepend(List, int);  
int get(List, int);  
void remove(List, int);
```

```
// list.c  
#include "list.h"  
struct link {  
    int value;  
    struct link *next;  
};  
struct list {  
    struct link *first;  
    struct link *last;  
};  
  
List mkList() {  
    List result = malloc(sizeof(struct list));  
    ...
```

Interfacet implementerat som en länkad lista



```
// list.h  
typedef struct list *List;  
  
List mkList();  
void append(List, int);  
void prepend(List, int);  
int get(List, int);  
void remove(List, int);
```

```
// list.c  
#include "list.h"  
struct list {  
    int values[256];  
    int largestIndex;  
};  
  
List mkList() {  
    List result = malloc(sizeof(struct list));  
    ...
```

Interfacet implementerat som en array



```
void prepend(List list, int value) {  
    list->first = mkLink(list->first, value);  
    if (list->last == NULL) {  
        list->last = list->first;  
    }  
}
```

O(1)

```
void prepend(List list, int value) {  
    if (list->largestIndex > 0) {  
        for (int i=largestIndex; i>0; --i)  
            list->values[i] = list->values[i-1];  
    }  
    list->values[0] = value;  
}
```

O(n)

Prepend för länkad lista och arraylista har samma gränssnitt men  
har väsentligt annorlunda implementation



# Inkapsling

---

- Teknik för att dölja implementationsdetaljer för utomstående
  - Distinktionen publika / privata funktioner och data
  - Inga beroenden av externt data som kan ändras godtyckligt
  - Kopiering, ägarskapstyper
- Vissa programspråk har explicit stöd för inkapsling genom kontroll av hur/var delar av deklarationer får användas (dock ej C)

```
// list.h

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

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

```
// myprog.c
#include "list.h"

void doubleinsert(List list, int v1, int v2) {
    list->first = mkLink(v1, mkLink(v2, list->first));
}
```



Undermålig inkapsling möjliggör hög coupling och dålig modularisering

```
// list.h
struct list {
    int values[256];
    ...
};
```

Byte till array-lista...

```
// myprog.c
#include "list.h"

void doubleinsert(List list, int v1, int v2) {
    list->first = mkLink(v1, mkLink(v2, list->first));
}
```



```
// myprog.c  
#include "list.h"  
  
void doubleinsert(List list, int v1, int v2) {  
    prepend(v1);  
    prepend(v2);  
}
```

Bättre och förändringssäker implementation.

Framtvingas av korrekt genomförd inkapsling.



# 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)