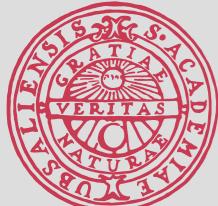


Abstraktion, modularisering och informationsgömning

Tobias Wrigstad
tobias.wrigstad@it.uu.se



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
- 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.
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

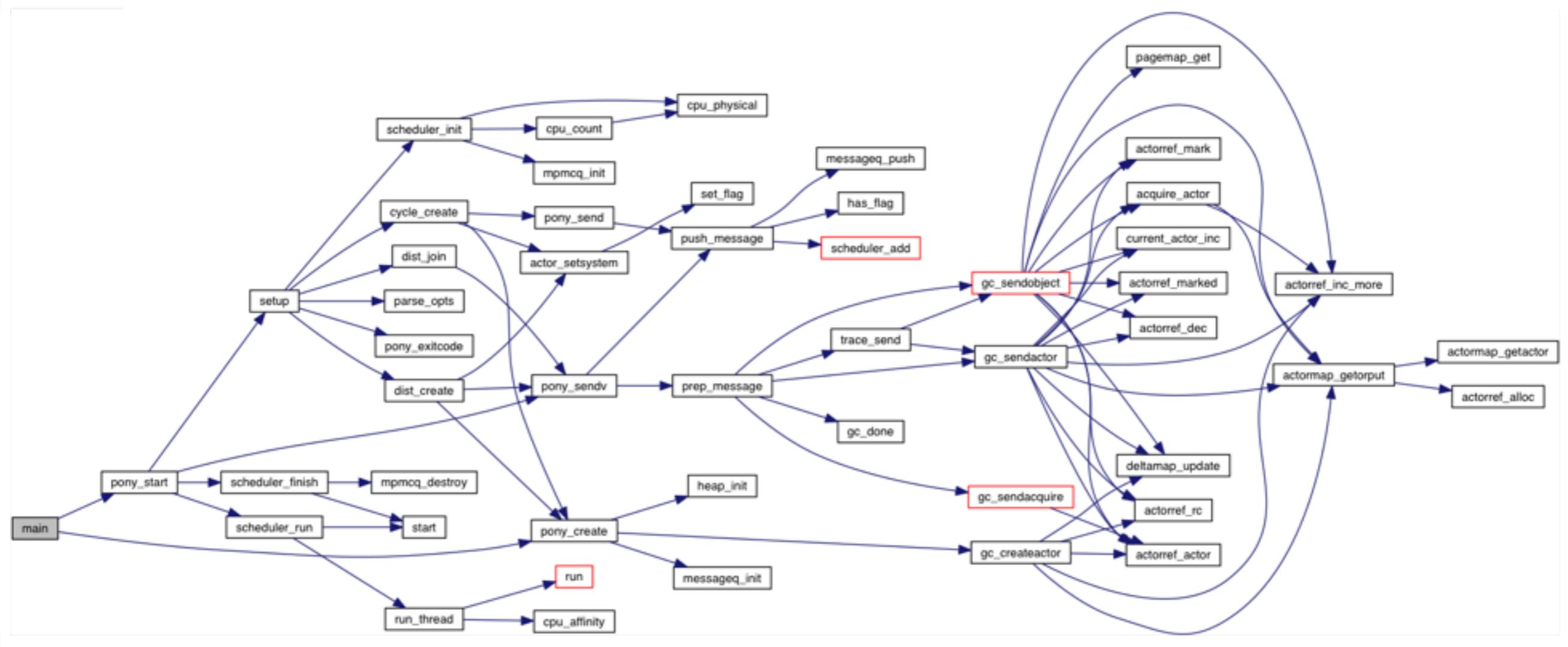
fermentum lacus.
Vestibulum quam.
Duis porta,
nibh
sed
congue
tincidunt,
risus
felis
porttitor orci,
vitae pharetra erat arcu eu
dolor.
Sed lacus nulla, auctor eget,
interdum eget, tempus sed,
pede.

saveToSentFolder

Ut odio mauris,
tincidunt ac,
molestie eu,
sagittis at,
wisi.
Praesent
lacinia
tellus ac
orci
laoreet
tincidunt.
Phasellus dictum



Kontrollabstraktion ger översikt

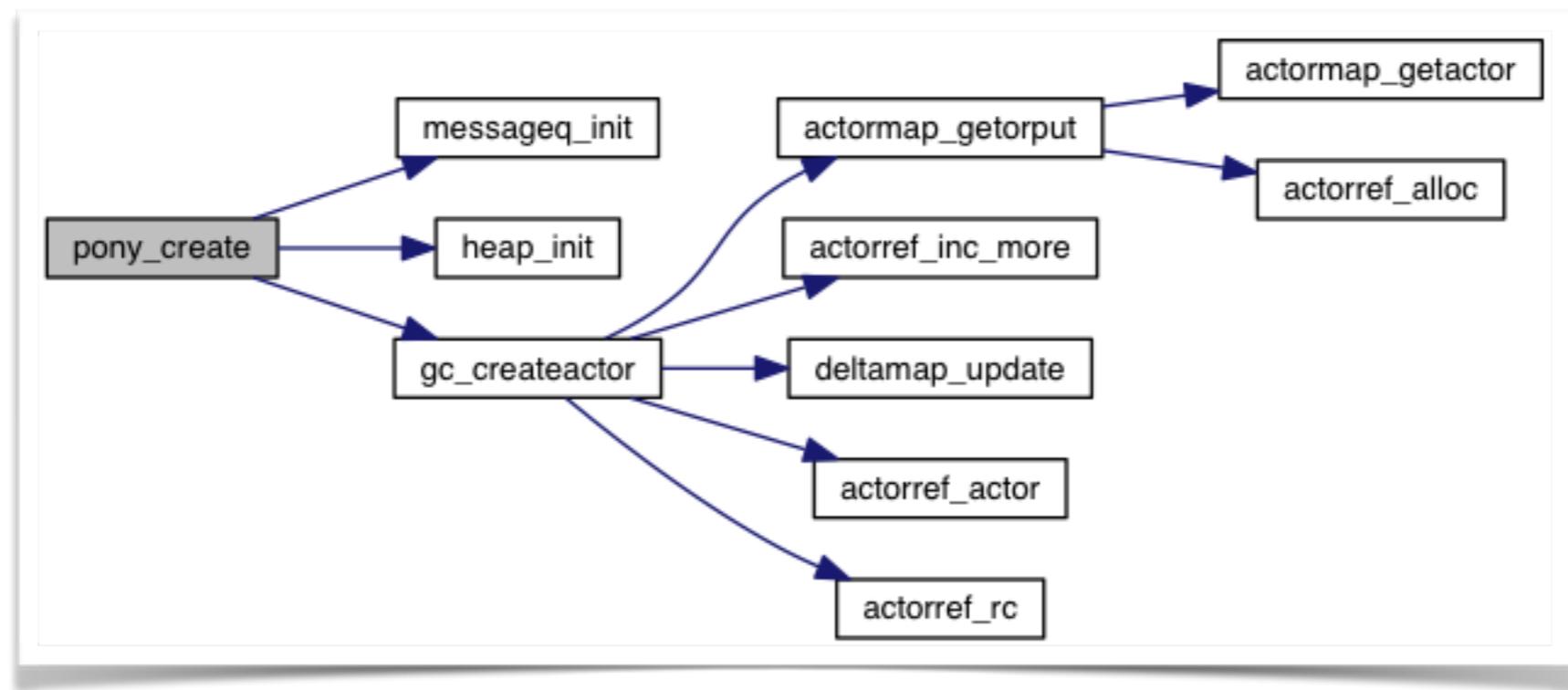


Anropsgraf

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



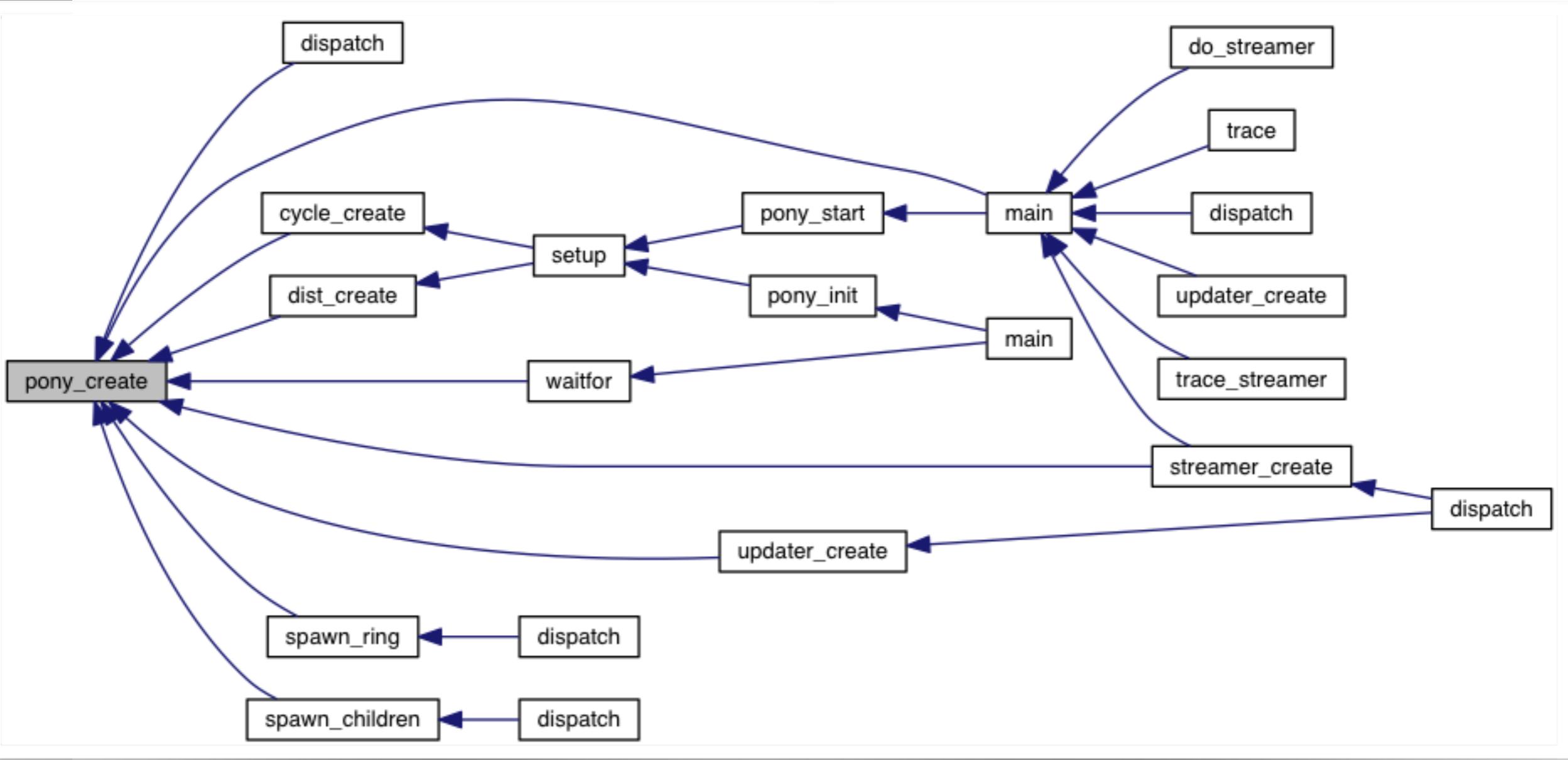
Kontrollabstraktion ger översikt



Anropsgraf rotad i `pony_create`



Kontrollabstraktion ger översikt



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 obekant för program som använder datat

Använd **accessorer** och **mutatorer** för manipulera datat som är kopplat till 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

- Abstrakta data: "specifikation"

Kopplar loss en datastrukturs
klienter från datastrukturen!

- 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

- Struktur
Programmet har "abstrakta data"

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

referenterat 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: "specification"
- 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

- Struktur
Prog:

```
commodity_t *book = ...;
book->cost = 12.50;
int cost_of_book = book->cost;
```
- Programmet refererar "abstrakta data"
- Implementeringen är särskilt visuell

```
int cost_in_eurocent(commodity_t *com);
void set_cost_in_eurocent(commodity_t *com, int cost);
```

- 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

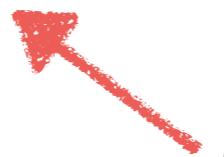
Kapsla in implementationsdetaljer och undvik externa beroenden



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

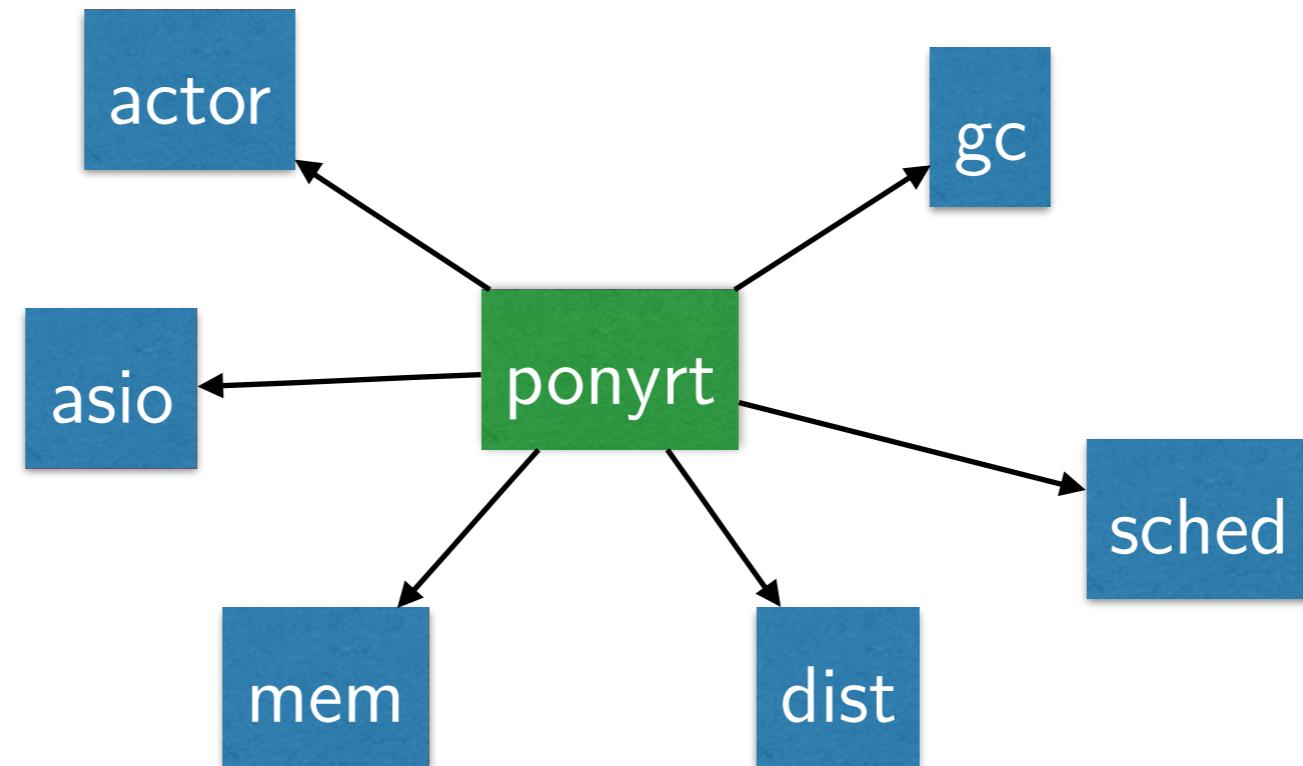


Modularisering

- 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 &
 - 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



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



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

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(...);
```

```
#include "list.h"  
  
void append(...){  
    ...  
}  
int length(...){  
    ...  
}  
int empty(...){  
    ...  
}  
List mkList(){  
    ...  
}  
struct link* mkLink(...){  
    ...  
}
```

Deklarationer &
funktionsprototyper

"Själva koden"



UU

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;
```

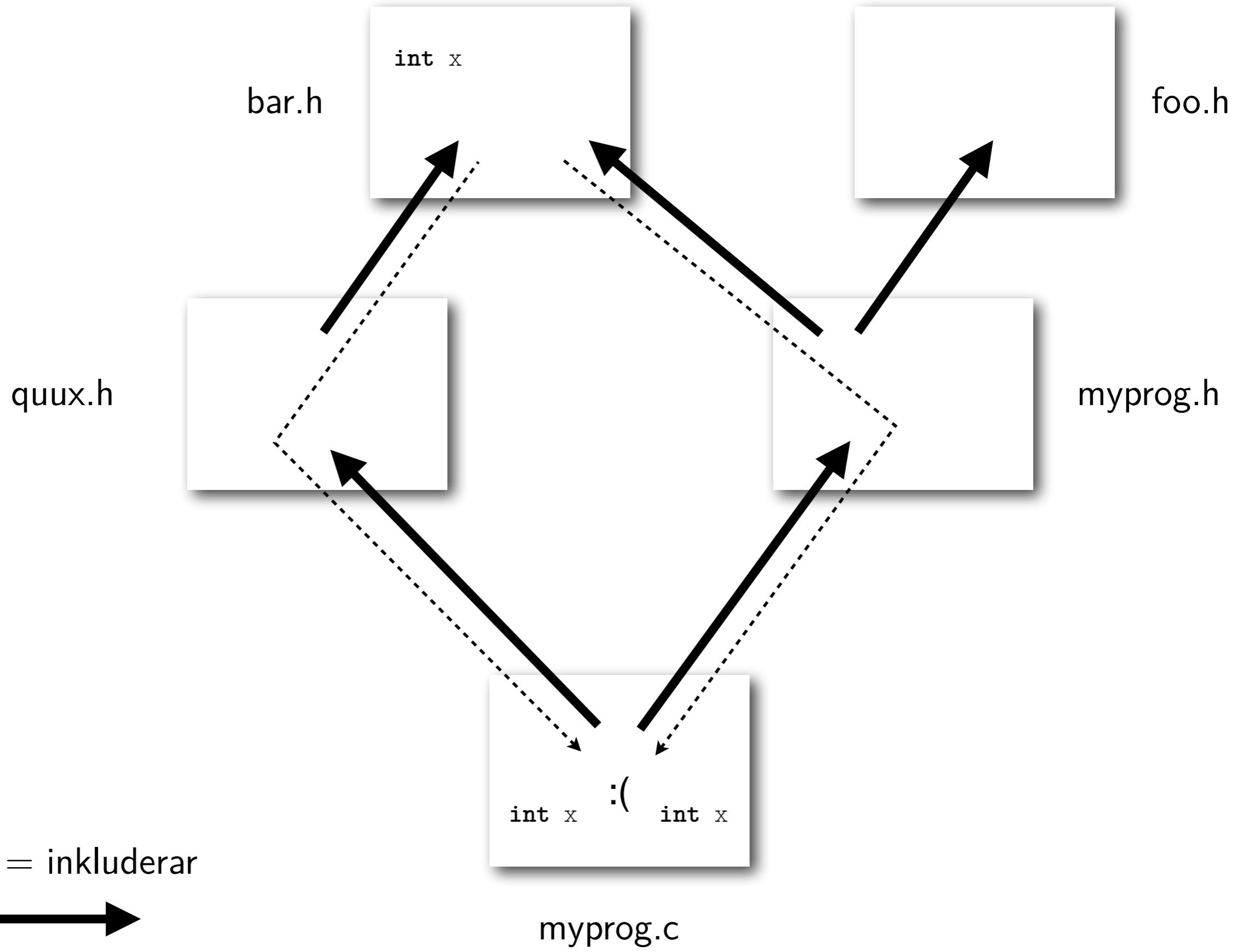
```
void append(List, int);  
int length(List);  
int empty(List);  
List mkList();  
struct link* mkLink(...);
```

```
void append(...) {  
    ...  
}  
int length(...) {  
    ...  
}  
int empty(...) {
```

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



problem med flerfaldig inkopiering...



UU

list.h

```
#ifndef __list_h
#define __list_h

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

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

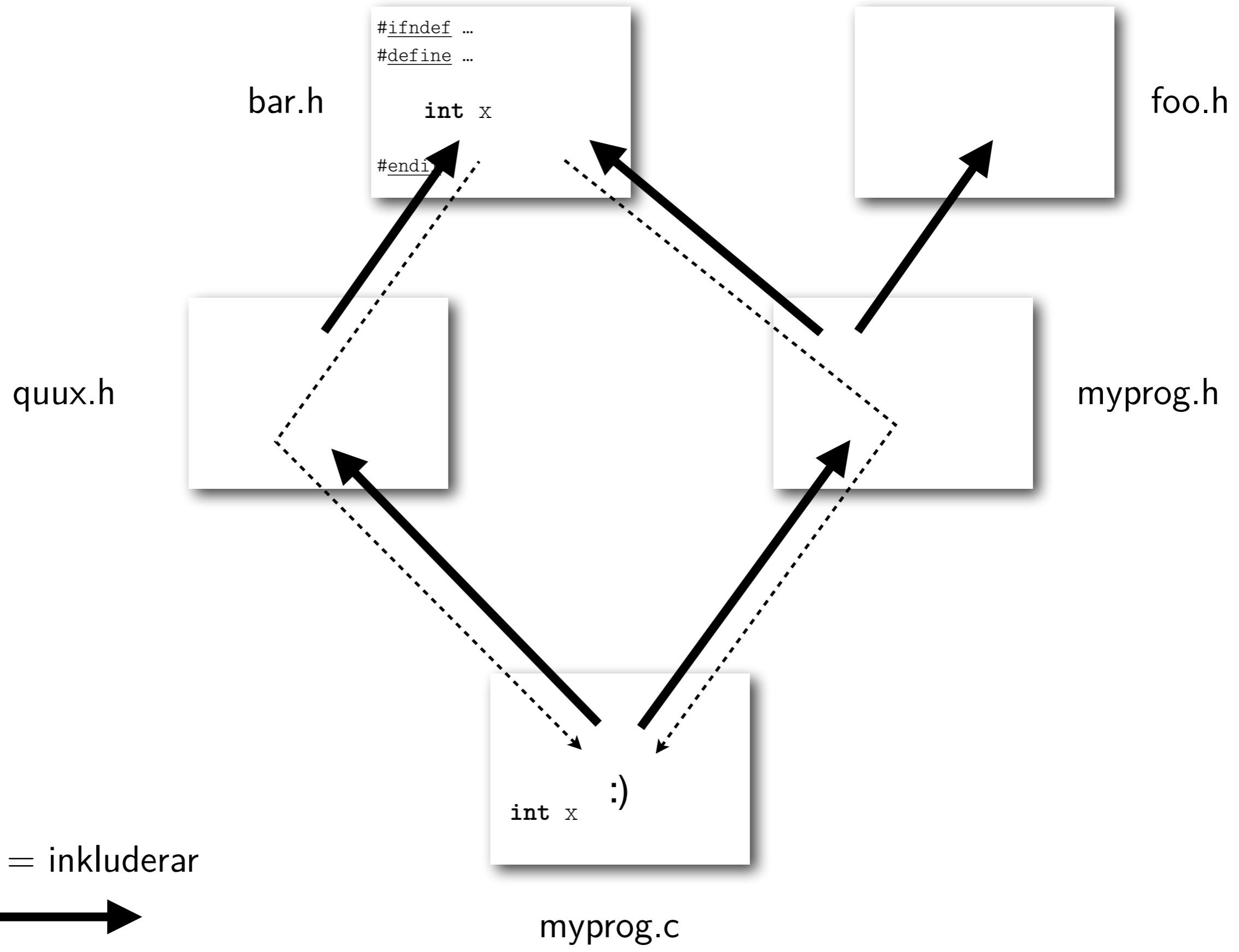
typedef struct list *List;

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

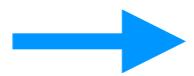
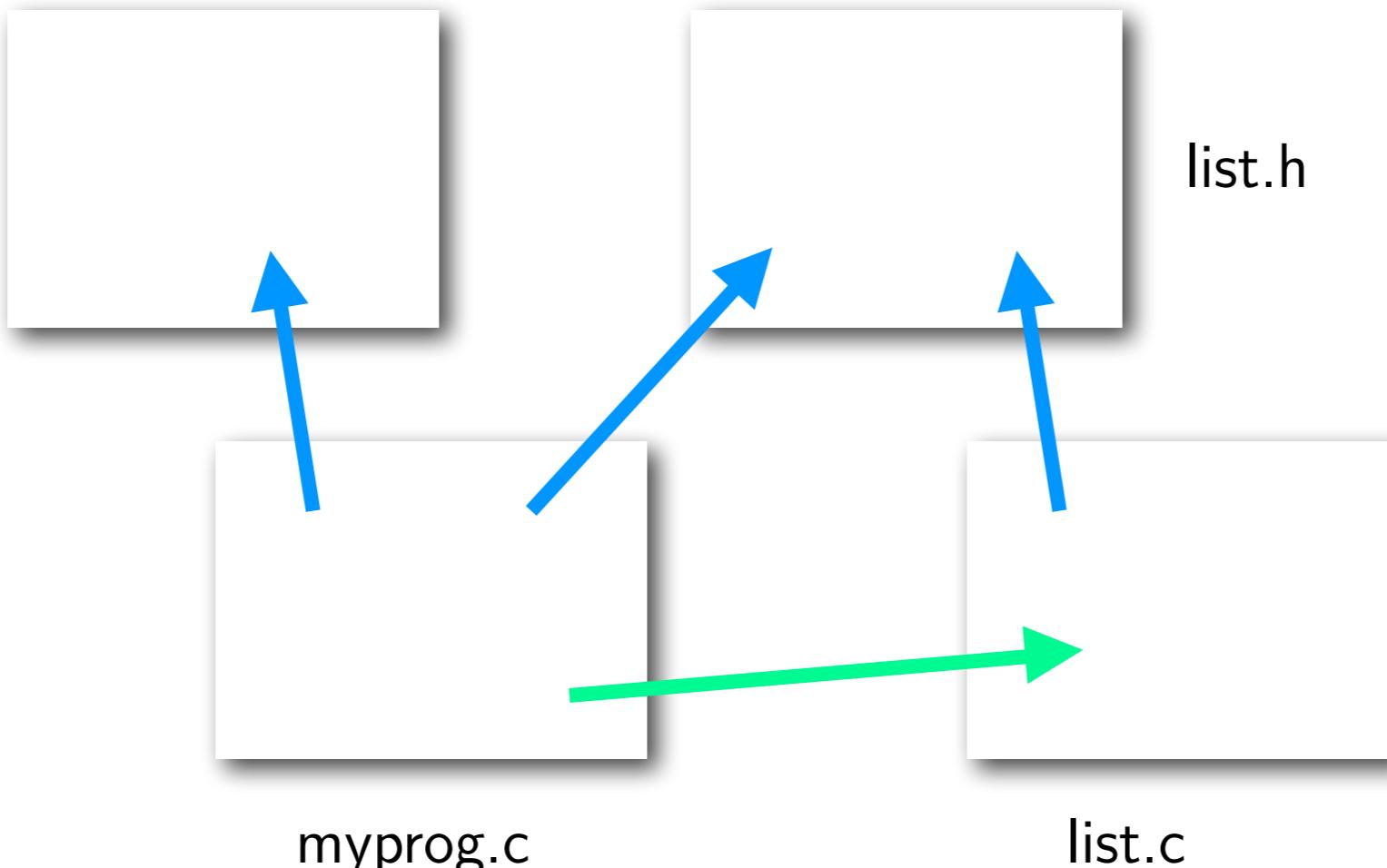
#endif
```

förhindrar flerfaldig inkopiering





myprog.h



= kompileringsberoende



= länkningsberoende



UU

Separatkompilering och länkning

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

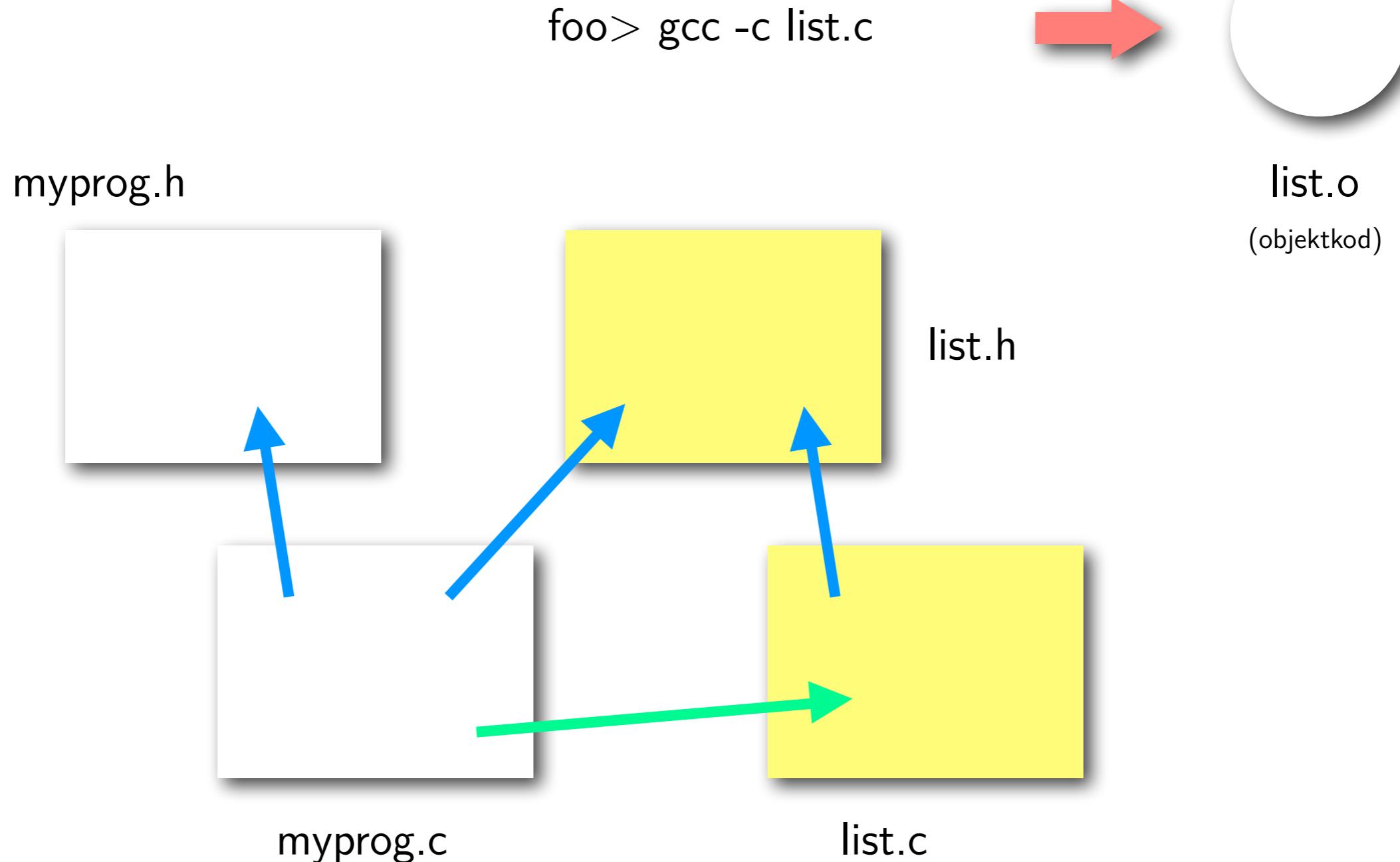
Kompilering medger statisk felkontroll

Objektkoden kan vidare distribueras

- Alla separatkompilerade moduler länkas slutligen ihop till ett körbart program

Länkningen löser ut beroenden mellan modulerna

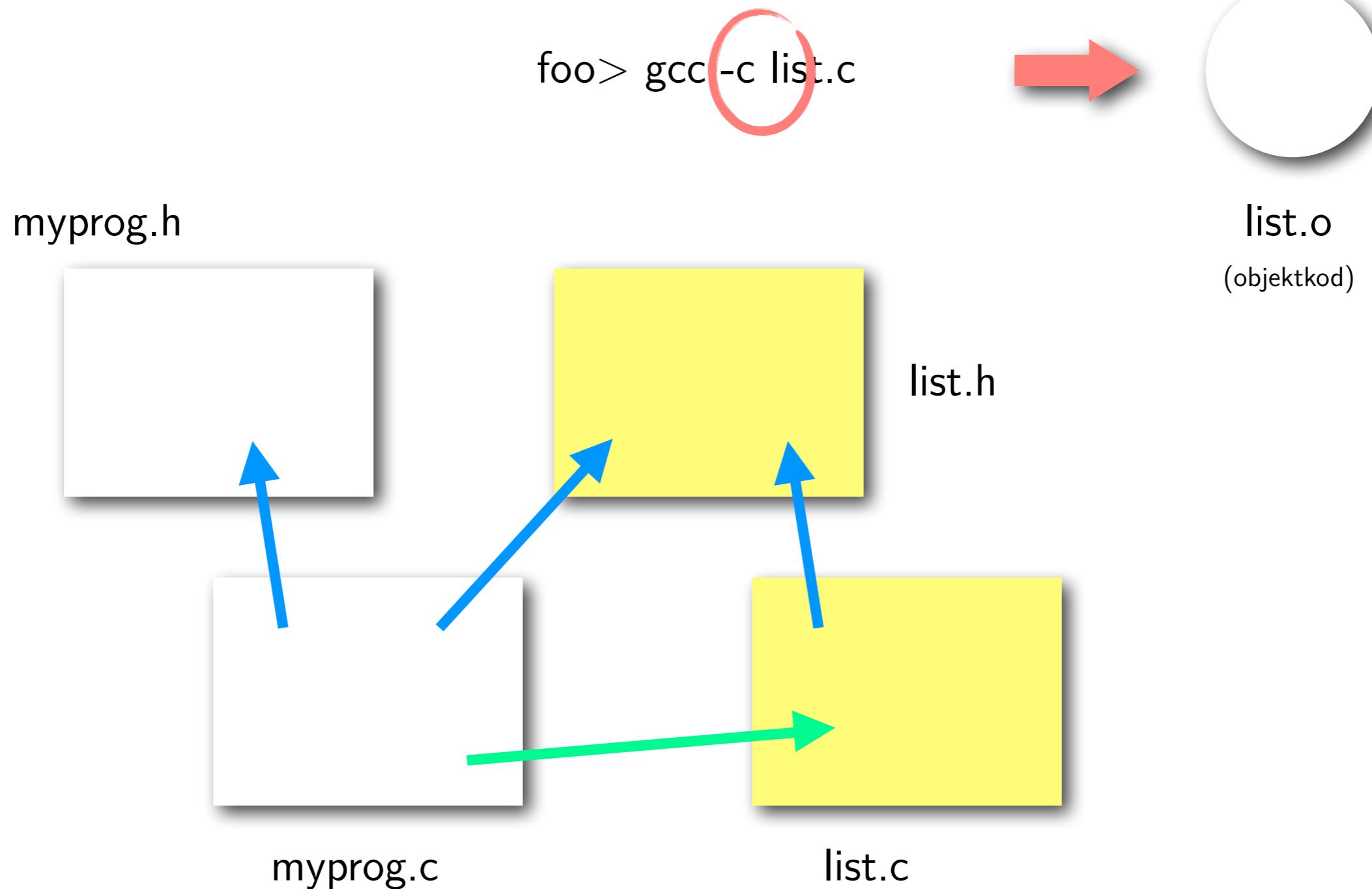




→ = kompileringsberoende

→ = länkningsberoende

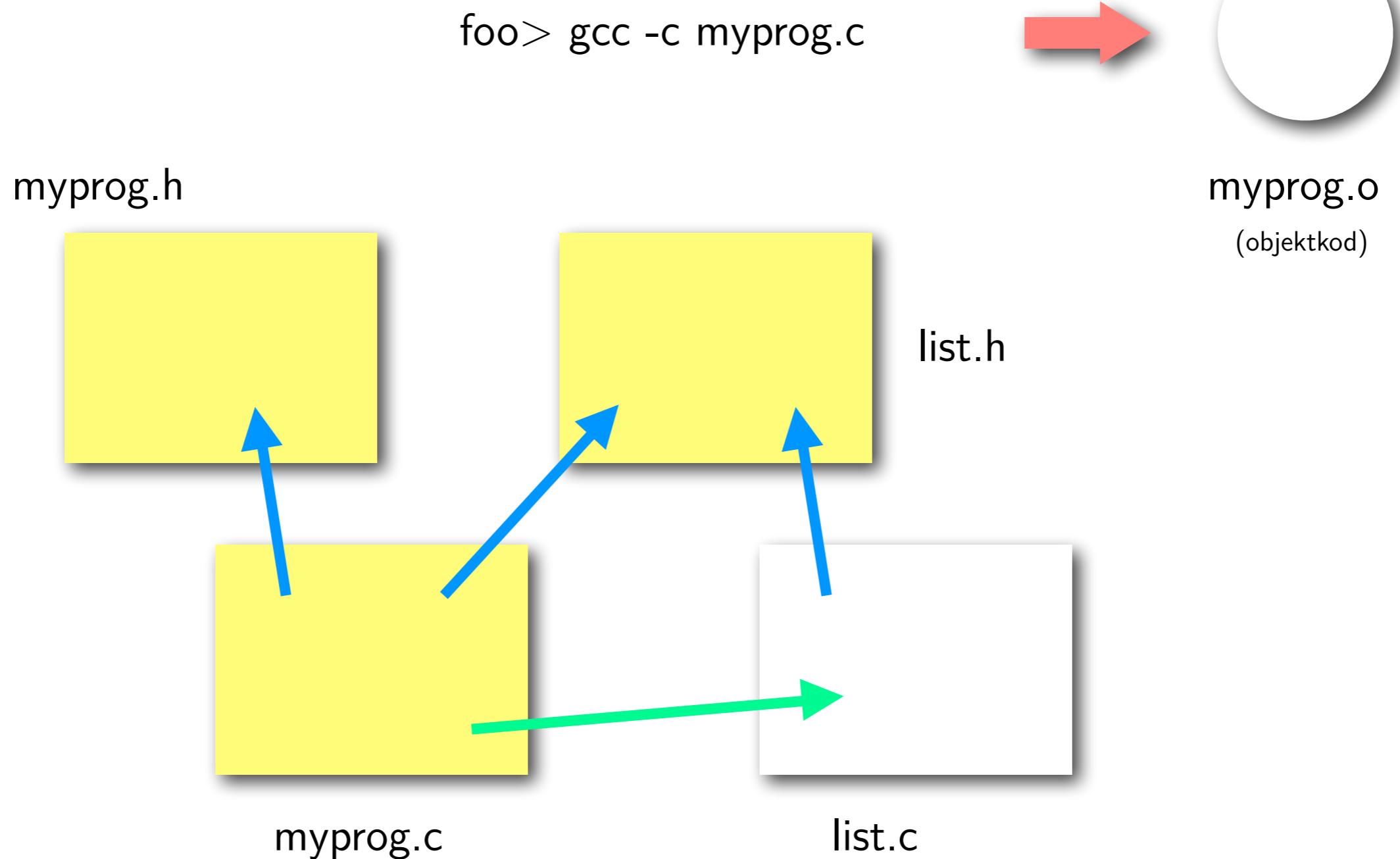




→ = kompileringsberoende

→ = länkningsberoende





→ = kompileringsberoende

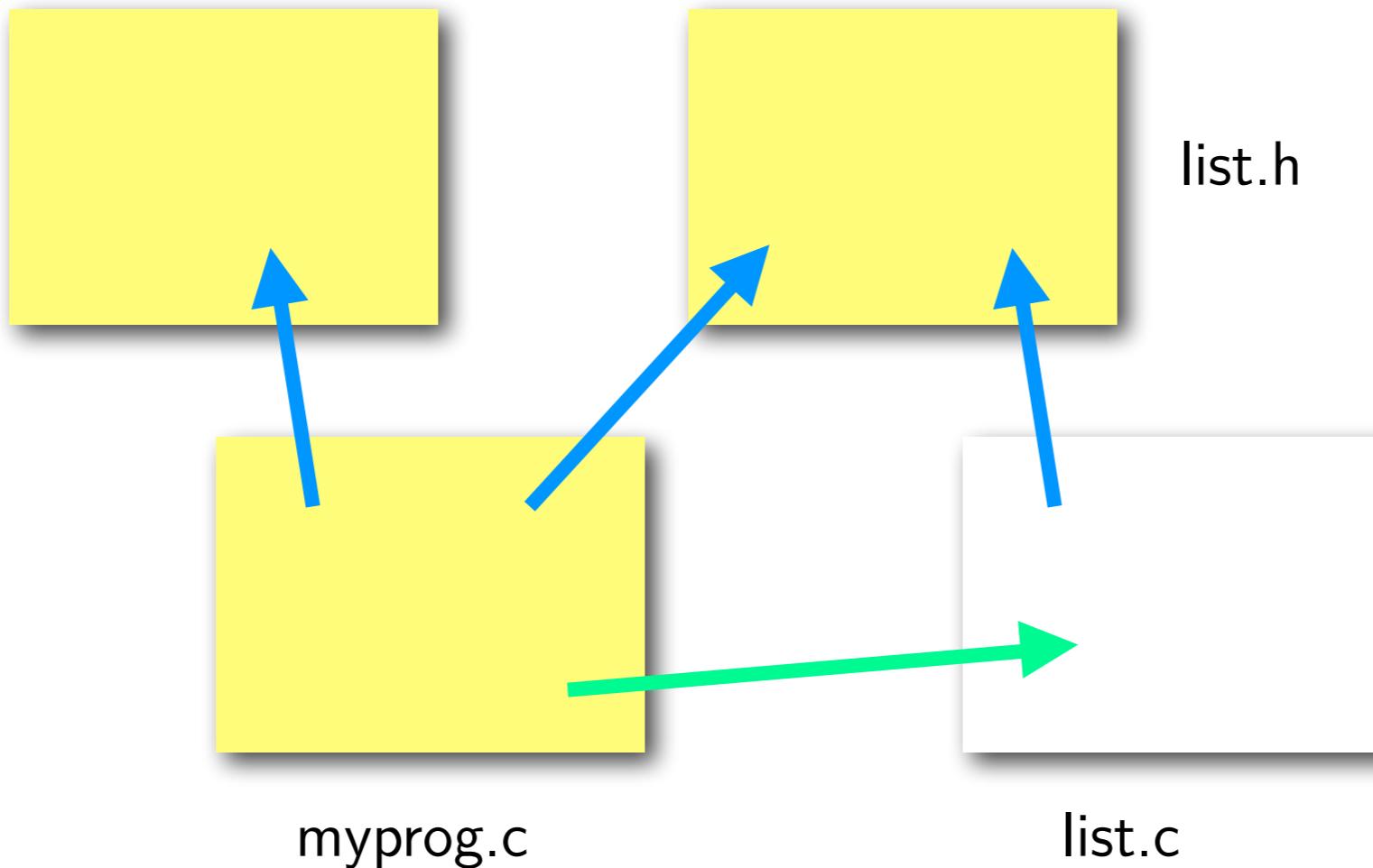
→ = länkningsberoende



foo> gcc myprog.c

kompilerar ej!

myprog.h



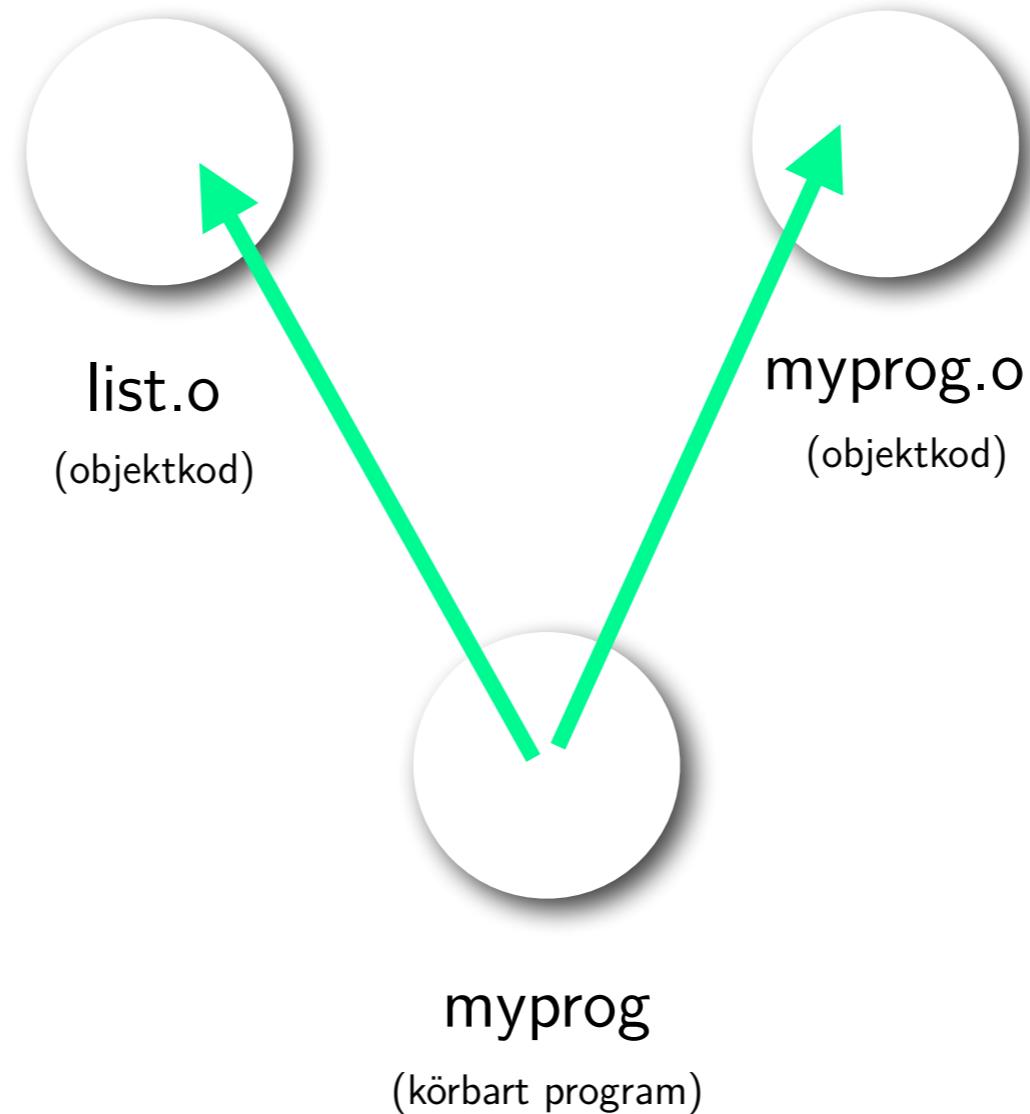
→ = kompileringsberoende

→ = länkningsberoende



UU

```
foo> 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) { ... }
```

logging

filhantering



UU

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



UU

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 (stabilia) 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"

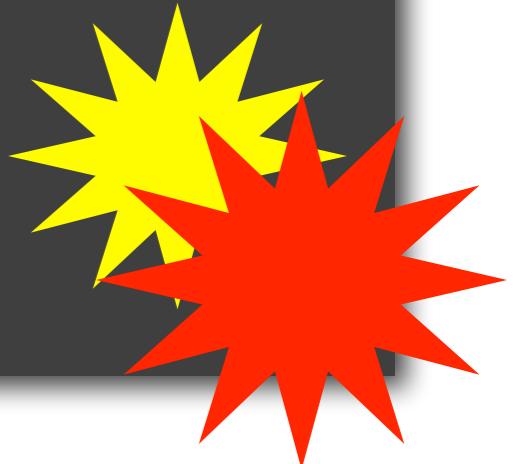
void doubleinsert(List list, int v1, int v2) {
    list->first = mkLink(v1, mkLink(v2, list->first));
}
```

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));  
}
```



UU

```
// 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)

