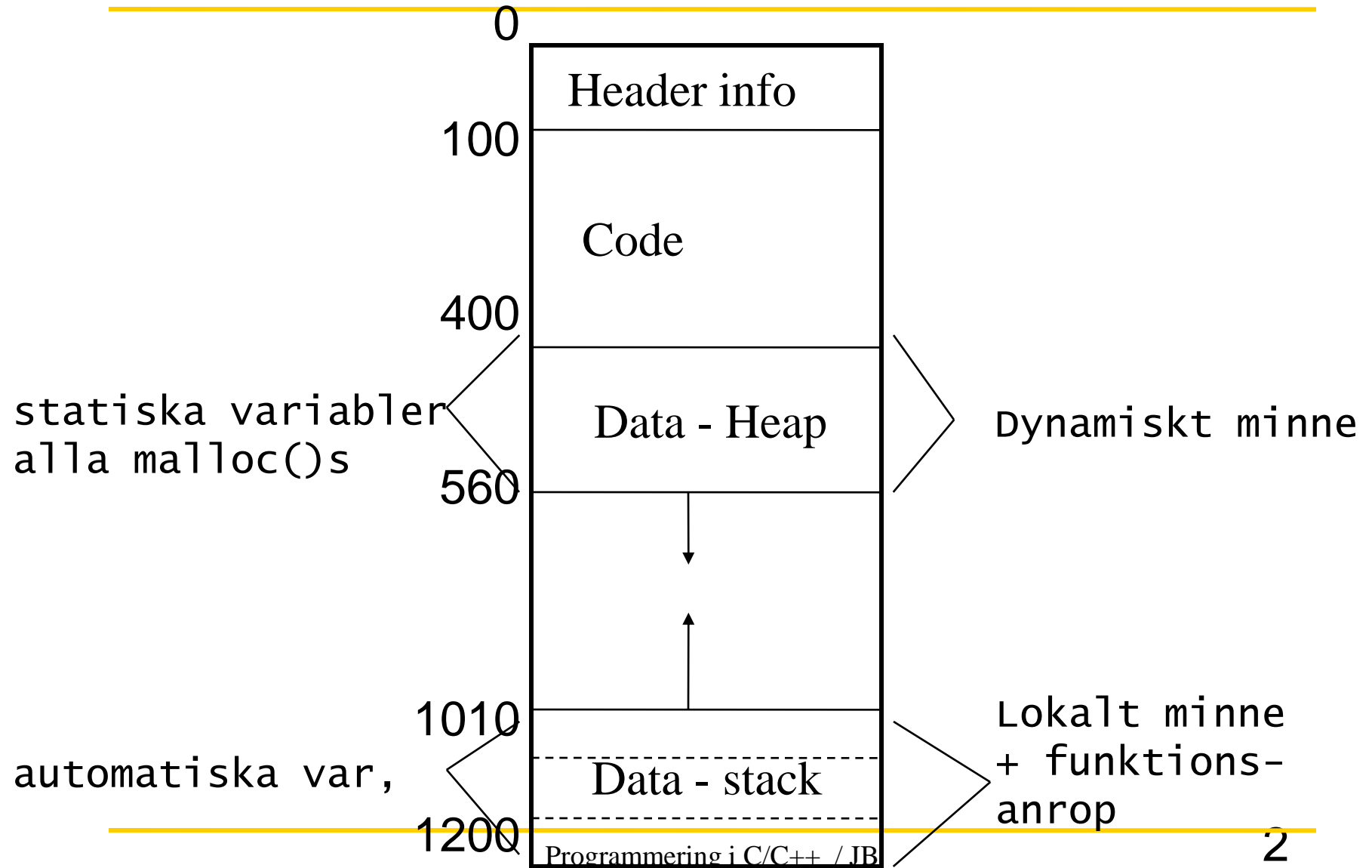

C-programmering

Minneshantering

Minnesrymd för program



Dynamisk minnesallokering

- Explicit allokering och deallokering

```
#include <stdio.h>

int main(void) {
    int *ptr;
        /* allokera utrymme för ett heltal */
    ptr = malloc(sizeof(int));

        /* används minnesutrymmet */
    *ptr=4;

    free(ptr);
        /* frigör det allokerade minnet */
}
```

Dynamiskt minne - funktioner

```
#include <malloc.h>
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *free(void *ptr);
void *realloc(void *ptr, size_t size);
```

`malloc` - returnerar pekare till minnesblock av storlek `size`

`realloc` - ökande / minskade av minne, gemensamt om oförändrad

`free` - frigör minne på pekare `ptr`

Dynamisk minnesallokering - exempel

```
#include <stdio.h>
#include <malloc.h>

int main() {
    char *buffer;
    pPerson ptr_pers;

    buffer = (char *)malloc(50 * sizeof(char));
    ptr_pers = (pPerson)malloc(40 * sizeof(Person));

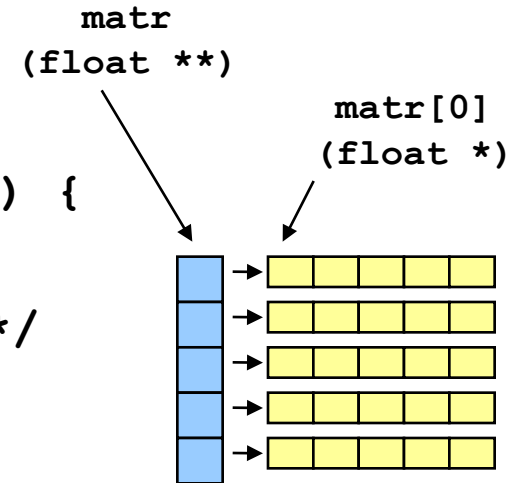
    if (!buffer) return (-1); /* om malloc ej lyckades */
    if (!ptr_pers) return (-1);
    strcpy(buffer, "Hej, här är texten");
    strcpy(ptr_pers[0]->namn, "Axel Eklund");
    ptr_pers[0]->alder = 32;
    return 0;
}
```

OBS!! fält i strukturer adresseras med `->` då pekare till struktur används

Dynamiska matriser

```
#include <stdio.h>
#include <malloc.h>

float ** matrix_alloc(int rows, int cols) {
    float **matr;
    int i;
    /* Allokera pekare till kolumnränder */
    matr = (float **)malloc(sizeof(float
    *)*rows);
    for(i=0; i<rows; i++) {
        /* Allokera kolumnränderna */
        matr[i] = (float
        *)malloc(sizeof(float)*cols);
    }
    return matr;
}
användning: matr[2][4] = 12.22;
```



Minnesläckor

- Allokerat minne (`malloc()`, `realloc()`, `calloc()`) har inte ett motsvarande `free()`
- Med tiden växer den allokerade minnesmängden
 - I serverbruk med tiden katastrofalt
 - För klientprogram (typ webbläsare) främst störande
- Hur kan man undvika
 - Allokering / deallokering på välkontrollerade platser
 - Noggrann felhantering (så att inte `free()` blir bortglömt p.g.a. ovanligt programflöde)
 - Verktyg vid utvecklingskedet

Minnesläckor - exempel

```
#include <stdio.h>
#include <malloc.h>

int myfunc() {
    char *buffer;

    buffer = (char *)malloc(50 * sizeof(char));
    if (!buffer) exit(-1); /* om malloc ej lyckades */
    scanf("%d", &newval);
    if (newval<0) return -1; /* error */
    ...
    free(buffer); /* programflödet når inte alltid
hit = möjlig minnesläcka */
}
```

C-programmering

libc i korthet

Filfunktioner i libc <-> systemet

- UNIX (Linux)-varianter använder internt i OS heltal för att identifiera filer <int>
- libc använder internt en datastruktur (FILE) för att buffra operationer till filer

```
libc: size_t fread( void *ptr, size_t size,  
    size_t nmem, FILE *stream );
```

```
OS: ssize_t read(int fd, void *buf, size_t  
    count);
```

```
fopen(), fread(), fclose(), fwrite(), fseek(),  
    ftell()
```

Sträng-hantering

- Prototyper för sträng hantering i `string.h`
- Funktioner för att
 - Beräkna längd på sträng (`strlen`)
 - Kopiera och förena strängar (`strcpy`, `strcat`)
 - Jämföra strängar (`strcmp`)
 - Söka i strängar (`strchr`, `strrchr`, `strcspn`)
 - Konvertering av strängar till numeriska värden (`atoi`, `atol`, `atof`)
 - Test för specifika värden (`isalnum`, `isalpha`, `isdigit`, ...)

Stränghantering - exempel

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

main () {
    char str1[100] = "Testing string handling";
    char str2[100];
    char *p;

    int i;
    printf("Längd sträng1: %u\n", strlen(str1));
    strcpy(str2, str1);
    /* ~Ekvivalent med */
    for (i=0; i<=strlen(str1);i++) str2[i] = str1[i];
    /* Men ... strängbiblioteket använder sig t.ex. av processorns
       Single-Instruction-Multiple-Data (SIMD) egenskaper */

    p = strstr(str1, "hand");
    printf("hand hittades på position %u i '%s'\n", p-str1, str1);
    return = 0;
}
```

Matematiska funktioner

- Matematikbiblioteket används genom att
 - inkludera: `#include <math.h>`
 - länka med matematiska biblioteket `lm`
 - `gcc myfile.c -o myfile -lm`
 - exempel:

```
#include <math.h>
main() {
    double x;
    x = cos(0.32*2*M_PI);
}
```

Tidsfunktioner

Prototyper:

```
#include <time.h>
```

```
#include <sys/timeb.h>
```

- `time_t time(time_t *tloc)` – antal sekunder sedan 00:00:00 GMT, 1. Jan 1970
 - När inträffar "Y2K-buggen?"
 - `time_t` är typedef som long int
 - Om 32-bitars arkitektur $2^{32} / 2 / 365 / 24 / 60 / 60 = 68$ år → år 2038
- `int ftime(struct timeb *tp)` - Mera noggrann tidsfunktion

```
struct timeb {
```

```
    time_t    time; /* se ovan */
```

```
    unsigned short millitm; /* 1/1000 s interval */
```

```
    short     timezone; /* tidszon (i minuter) */
```

```
    short     dstflag; /* flagga för sommartid */
```

```
};
```

Tidsfunktioner (2)

- **char *ctime(time_t *clock)** – koverterar tid till ASCII-representation
- **char asctime(struct tm *tm)** – konverterar en tm-struktur till ASCII

```
struct tm {  
    int    tm_sec;        /* seconds */  
    int    tm_min;        /* minutes */  
    int    tm_hour;       /* hours */  
    int    tm_mday;       /* day of the month */  
    int    tm_mon;        /* month */  
    int    tm_year;       /* year */  
    int    tm_wday;       /* day of the week */  
    int    tm_yday;       /* day in the year */  
    int    tm_isdst;      /* daylight saving time  
*/  
};
```

Nätverk - funktioner

- BSD Socket
 - Ändpunkt för kommunikation
 - Fil som det går att skriva till / läsa från
 - Skapas med `socket()`
 - Kopplas med `connect()`
- Nätverk OBS. Host byte/bit order är kanske inte samma som på nätet
 - Funktioner som `htons()` – host to network byte order

Nätverk – läs från webserver

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

char msg[] = "GET / HTTP/1.0\n\n";
char retbuf[4096];

int main() {
    int sockfd;
    struct sockaddr_in raddr;
    int count;

    sockfd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

    raddr.sin_family = AF_INET;
    raddr.sin_port = htons(80);
    /* www.abo.fi = 130.232.213.59 */
    inet_pton(AF_INET, "130.232.213.59", &raddr.sin_addr);

    if (-1 == connect(sockfd, (struct sockaddr *)&raddr, sizeof (struct
sockaddr_
in))) {
        perror();
        exit(-1);
    }

    write(sockfd, msg, strlen(msg)+1);
    count = read(sockfd, retbuf, 4096);
    retbuf[count] = 0;
    printf(retbuf);
}
```

Nätverk – liten server

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

char msg[] = "GET / HTTP/1.0\n\n";
char retbuf[4096];

int main() {
    int sockfd, c_sock;

    struct sockaddr_in saddr;
    struct sockaddr_in caddr;
    int count;

    sockfd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(9876);
    saddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    bind(sockfd, (struct sockaddr *)&saddr, sizeof(saddr));

    listen(sockfd, 5);
    while (1) {
        char ch;
        socklen_t clen = sizeof(caddr);
        printf("Server waiting\n");

        c_sock = accept(sockfd, (struct sockaddr *)&caddr,
            &clen);
        printf("Connection from '%s'\n",
            inet_ntoa(&caddr.sin_addr.s_addr));

        read(c_sock, &ch, 1);
        ch++;
        write(c_sock, &ch, 1);
        close(c_sock);
    }
}
```

C-programmering

Preprocessorn

Preprocessor

- Kompilering av C kod föregås av en preprocessor
- Preprocessorn genererar en ny "fil" med källkod genom att
 - ersätta definierade textsträngar
 - inkludera andra filer
 - konditional inkludering av textavsnitt
- Denna nya enhetliga fil körs genom den egentliga C-kompilatorn

Preprocessor – exempel

```
/* test.c */
```

```
#define MAX(a,b) ( ((a)>(b)) ? (a) : (b) )
```

```
#ifdef DEBUG
```

```
#define DPRINT(x) printf("Debug: '%s\n",  
    x);
```

```
#else
```

```
#define DPRINT(x)
```

```
#endif
```

```
main () {
```

```
    int x = 33;
```

```
    int y = 12;
```

```
    printf("Max av 2 och 5 ar %i\n",  
        MAX(2,5));
```

```
    printf("Max av x och y ar %i\n",  
        MAX(x,y));
```

```
    DPRINT("Nu narmar sig slutet");
```

```
}
```

Notera formen
expr1 ? expr2 : expr3
evaluera expr1; Om olika noll, evaluera till expr2; annars till expr3.

```
main () {  
    int x = 33;  
    int y = 12;
```

```
    printf("Max av 2 och 5 ar %i\n", (  
        ((2)>(5)) ? (2) : (5) ));  
    printf("Max av x och y ar %i\n", (  
        ((x)>(y)) ? (x) : (y) ));
```

```
gcc -E test.c -DDEBUG
```

```
    printf("Debug: '%s\n", "Nu narmar sig  
    slutet");;
```

```
gcc -E test.c
```

```
main () {  
    int x = 33;  
    int y = 12;
```

```
    printf("Max av 2 och 5 ar %i\n", (  
        ((2)>(5)) ? (2) : (5) ));  
    printf("Max av x och y ar %i\n", (  
        ((x)>(y)) ? (x) : (y) ));
```

```
}
```

C Preprocessorn

- **#define** – Definiera / substitutionera
 - **#define MAX_ARRAYSIZE 100**
 - **#define DOUBLE(X) ((x)*2)**
- **#include <fil.h>** – Inkludera fil
 - Sök reda på fil och sätt filens innehåll på motsvarande plats
- **#if #elif #else #endif**
 - Inkludera textavsnitt om ett uttryck gäller – dynamisk kod
- **#ifdef #endif**
 - Inkludera kod om symbol definiera

C Preprocessorn

- Effektiv för
 - debuggning
 - ”konstanter” som är lätta att vid omkompilering ändra på
 - konditionella kodblock (typisk vid multiplattform / -arkitektur)
 - konfigurering av system
- ```
#ifdef USE_WLAN
 kod för WLAN-modul
#endif
```
- OBS! Ingen run-time overhead !!

# C Preprocessor – mera debugging

---

- De flesta preprocessorer erbjuder macron
  - `__FILE__` - ersätts med filnamnet för nuvarande fil
  - `__LINE__` - ersätts med radnumret i filen
  - `__DATE__` - datum då preprocessorern körs
- Vi kan nu skapa debug-macron

```
#define DPRINT(x) fprintf(stderr,
 "%s(%i): %s\n", __FILE__, __LINE__,
 (x))
```

- i koden:

```
DPRINT("Fel information given");
```

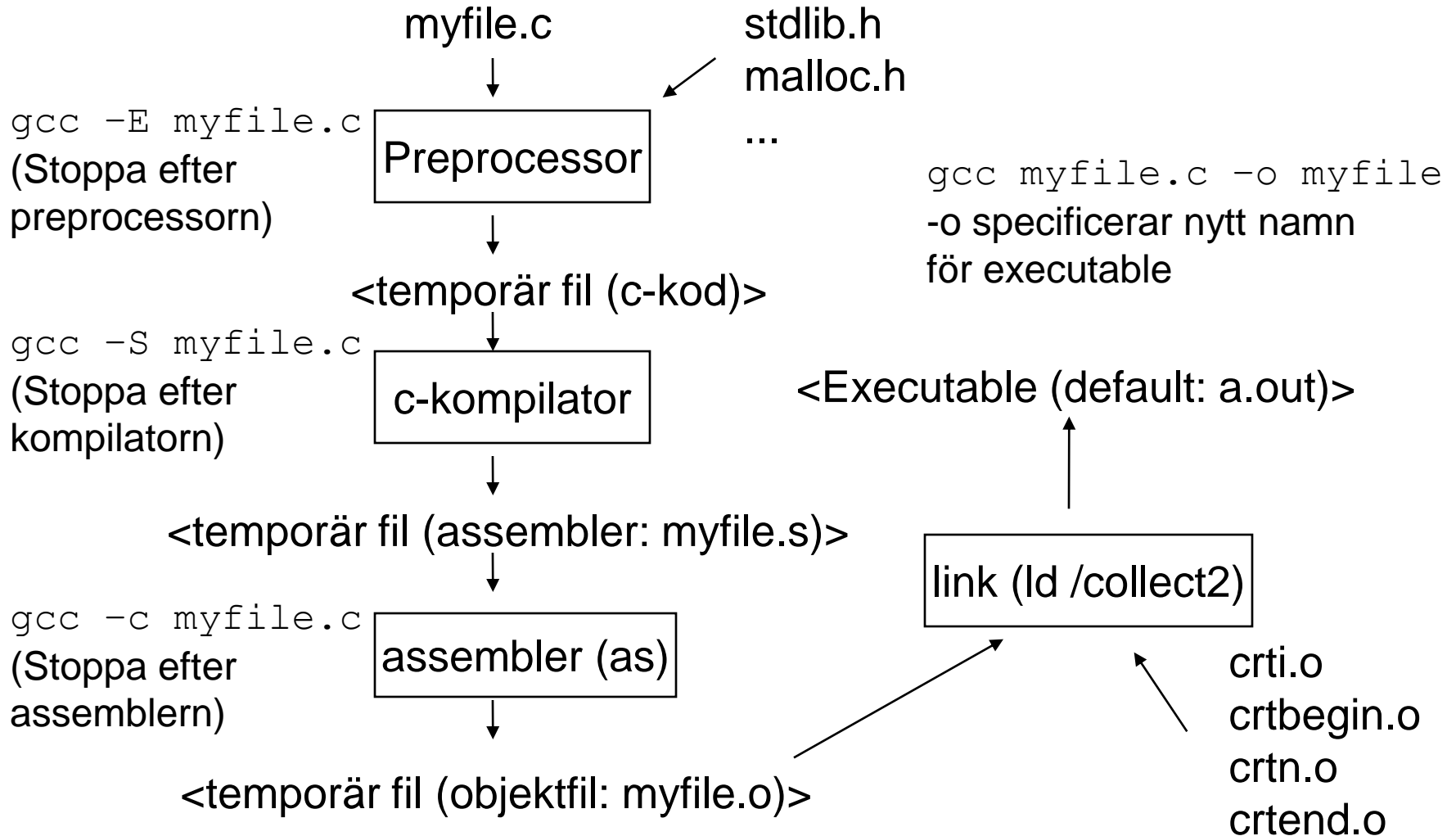
- Ger output

```
sourcefile.c(232): Fel information given
```

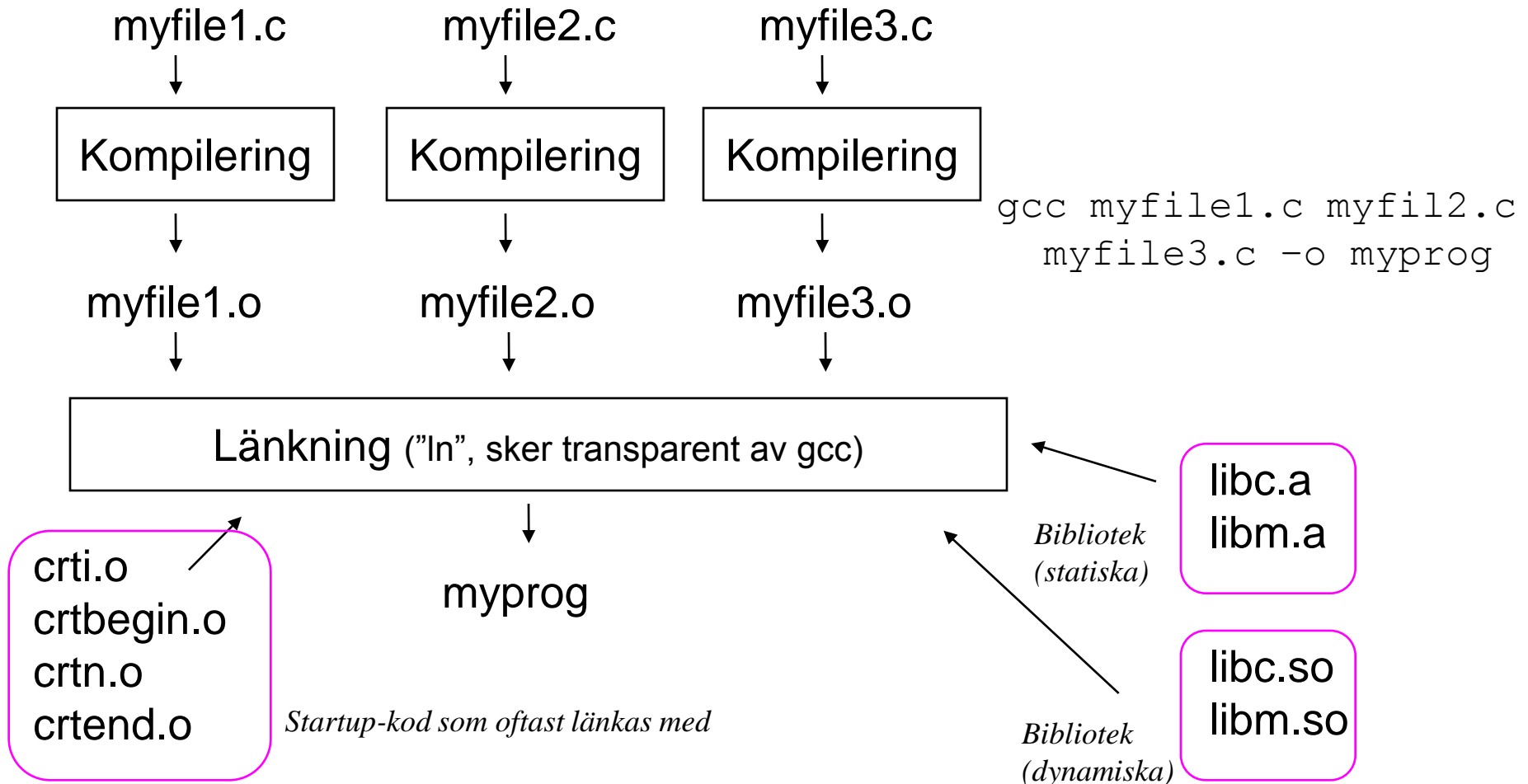


# Kompileringsprocessen

---



# Vanligen många källfiler



# Vad gör länkaren?

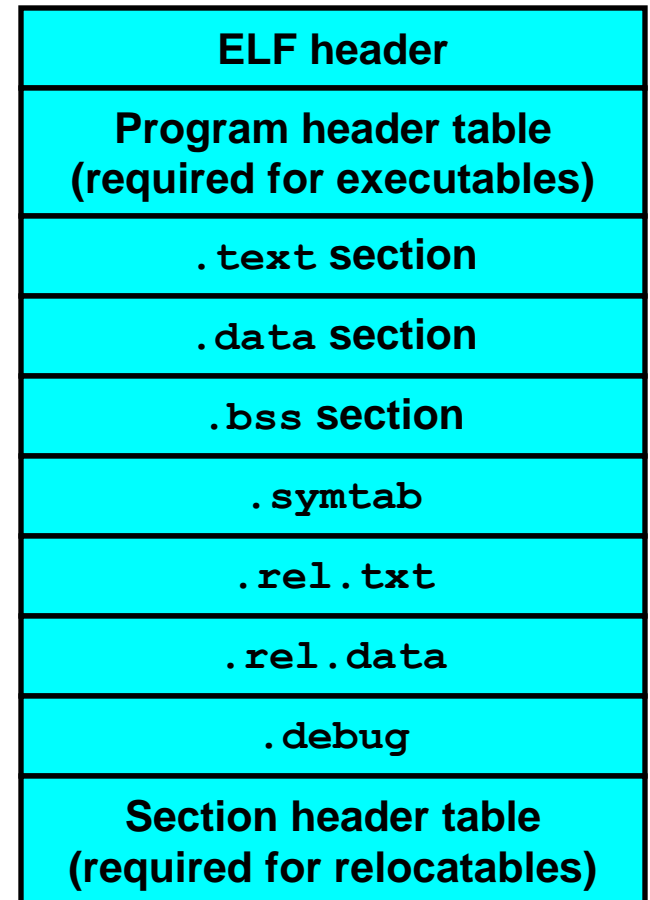
---

- Sammafogar filer med objektкод (.o)
  - Flyttbar objektкод kombineras till en exekverbar fil
- Externa referenser uppsöks
  - Referenser till funktioner och globala (externa referenser) mellan objektfiler
- Flyttar symboler
  - Flyttar symboler från relativa platser i objektfilerna till absoluta platser i den exekverbara filen
  - Uppdaterar alla referenser till dessa symboler
    - Referenser finns till både kod och data

```
a() /*referens till kodsymbol */
int *xp = &x; /* referens till datasymbol */
```

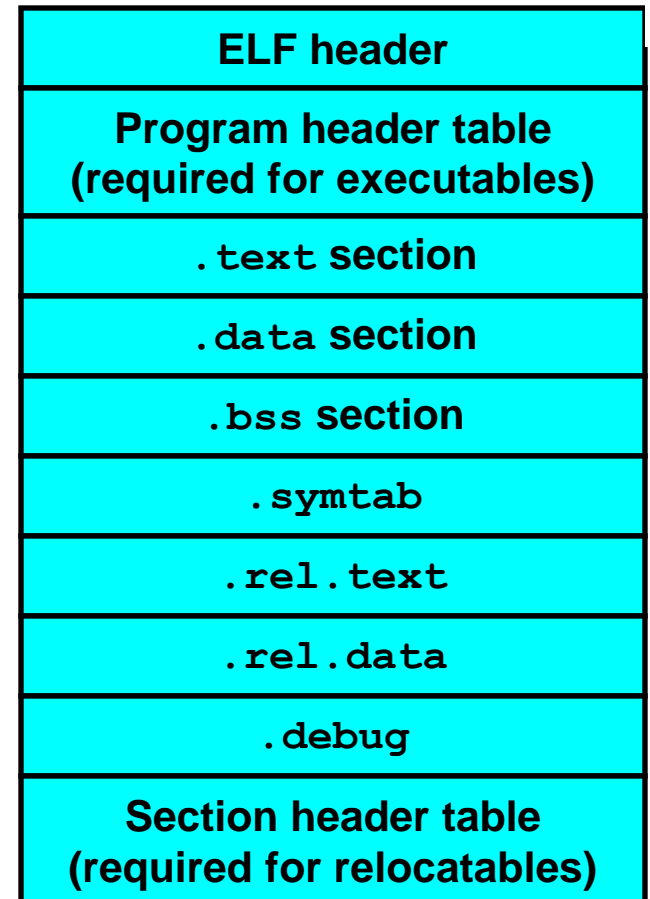
# ELF Object File Format

- Elf header
  - Magic number, type (.o, exec, .so), machine, byte ordering, etc.
- Program header table
  - Page size, virtual addresses memory segments (sections), segment sizes.
- .text section
  - Code
- .data section
  - Initialized (static) data
- .bss section
  - Uninitialized (static) data
  - “Block Started by Symbol”
  - **“Better Save Space”**
  - Has section header but occupies no space



# ELF Object File Format (cont)

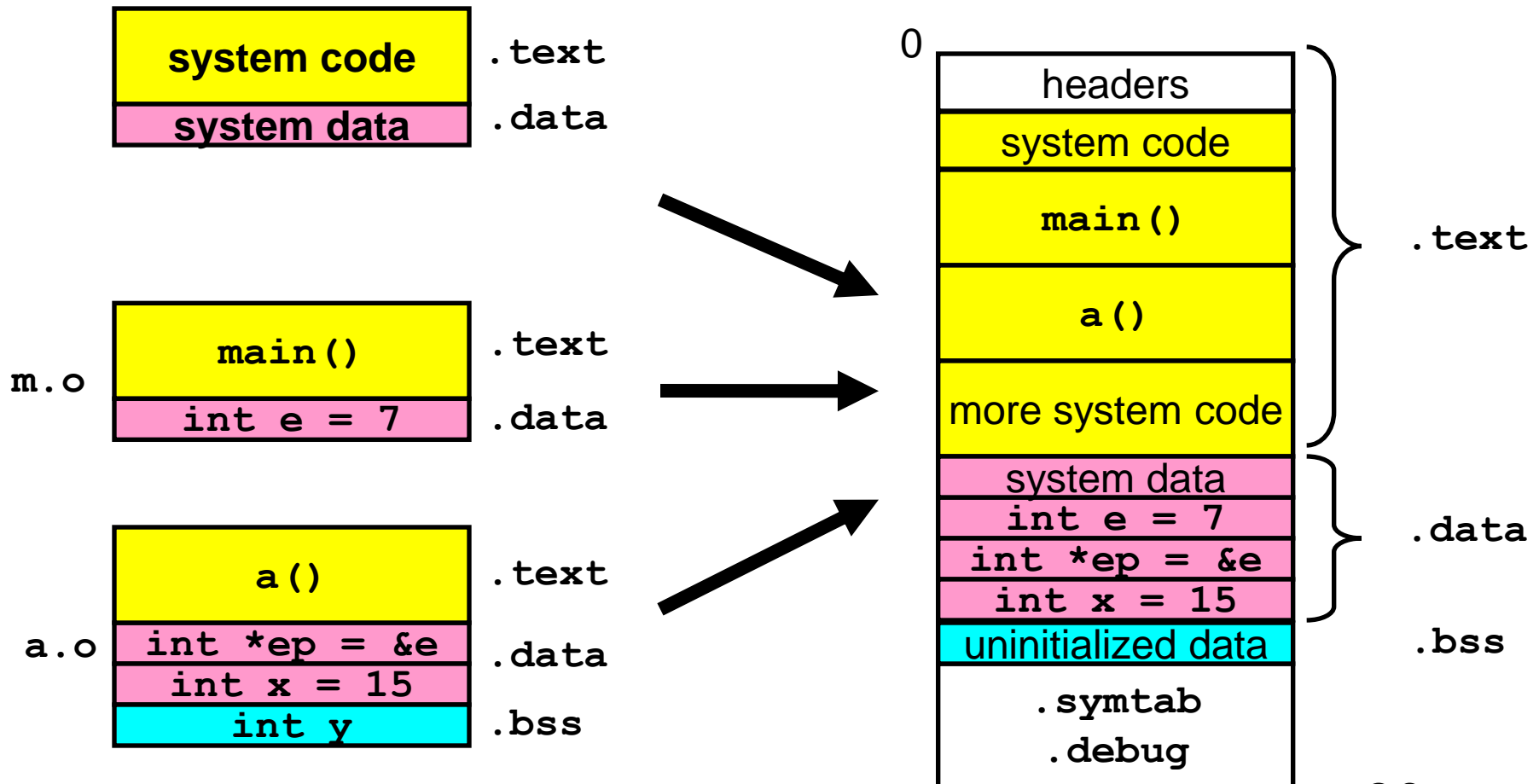
- `.symtab` section
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- `.rel.text` section
  - Relocation info for `.text` section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying.
- `.rel.data` section
  - Relocation info for `.data` section
  - Addresses of pointer data that will need to be modified in the merged executable
- `.debug` section
  - Info for symbolic debugging (`gcc -g`)



# Kombination av objektfiler till exekverbar fil

## Relocatable Object Files

## Executable Object File

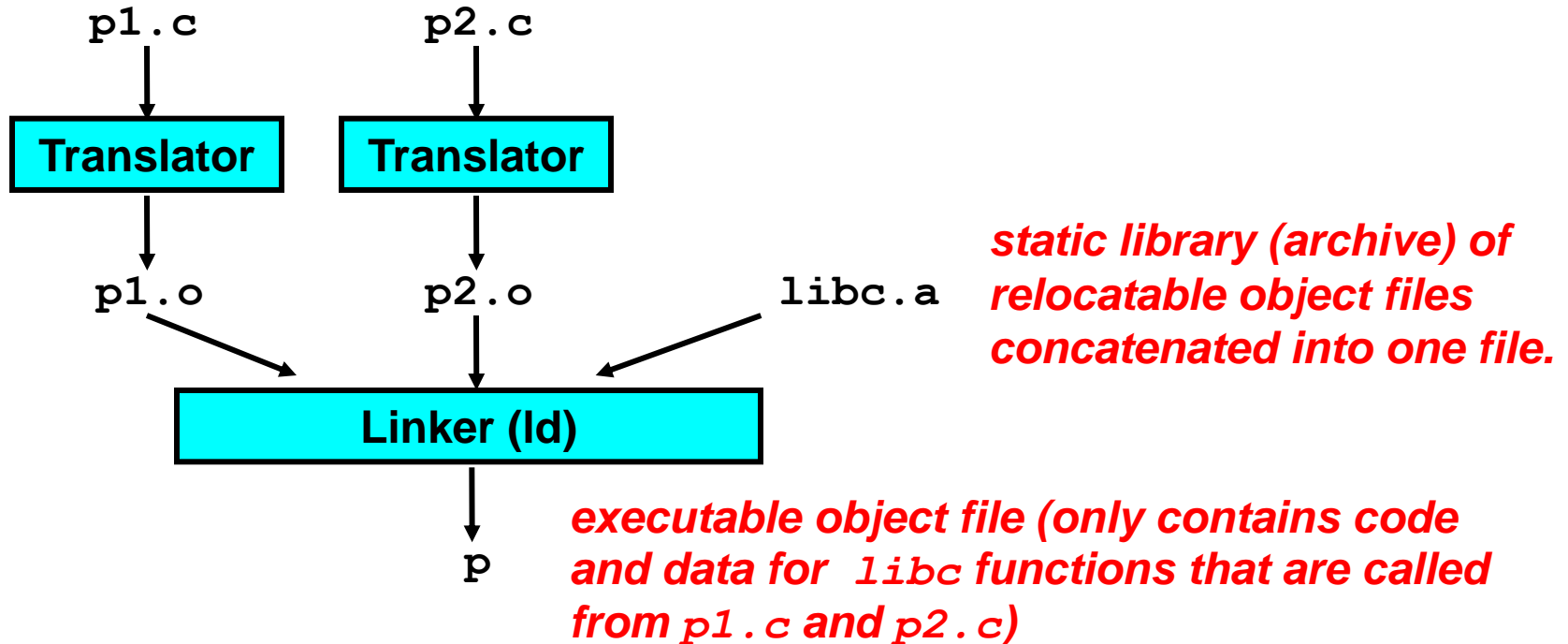


# Länkingsmodeller

---

- Statisk länkade exekverbara filer
  - Koderna för biblioteksfunktioner är inkluderad i den slutliga exekverbara filen
    - `gcc -static myfile.c -o myfile.c`
  - Exekverbara filer blir större
  - Dock bättre resistent mot ändringar i biblioteksfunktioner
- Dynamisk länkade exekverbara filer
  - Koderna för biblioteksfunktioner laddas vid körning från bibliotek
    - `gcc myfile.c -o myfile.c`
  - Mindre exekverbara filer
  - Kan dock bli problem med inkompatibla bibliotek

# Statisk länkning

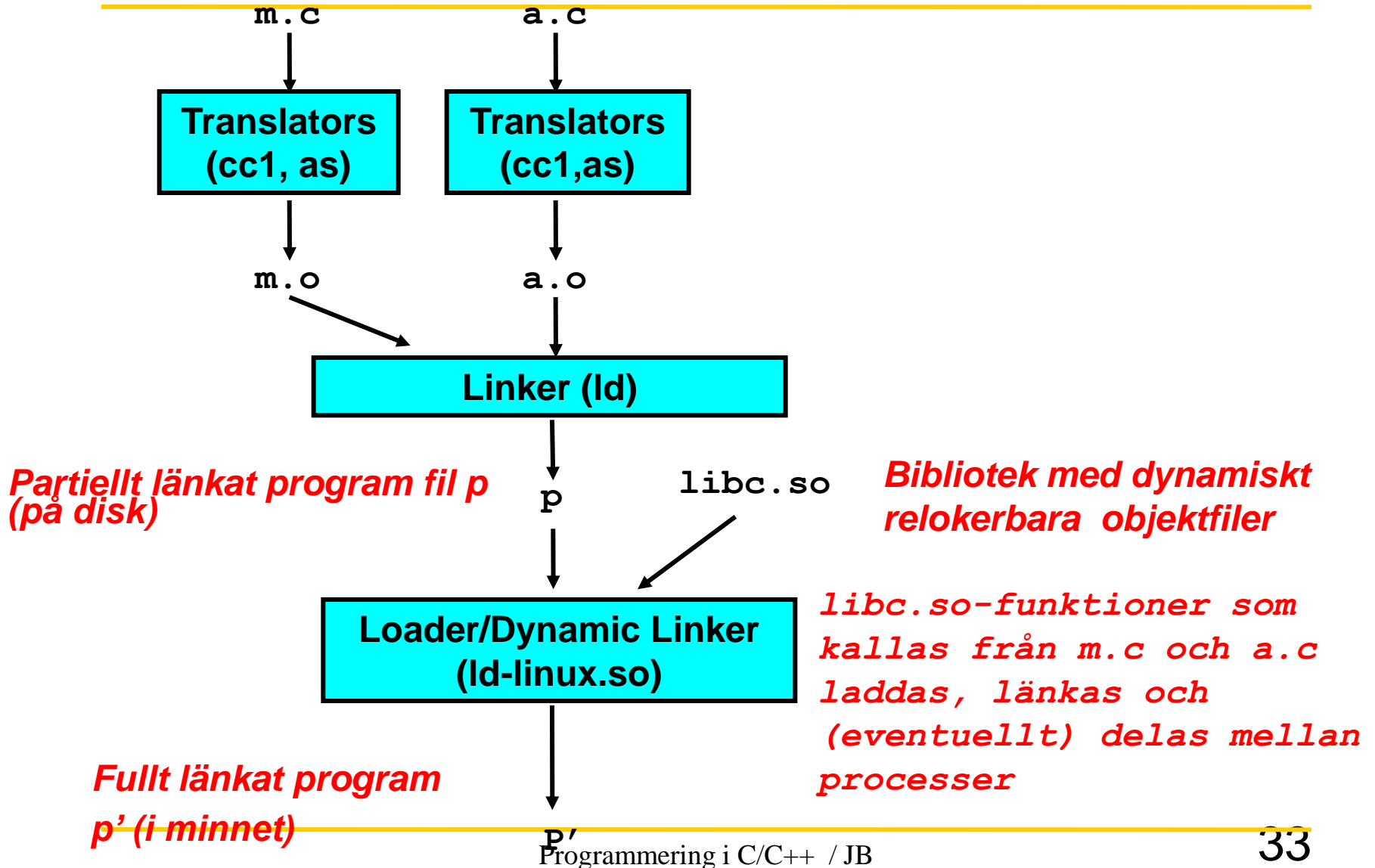


Further improves modularity and efficiency by packaging commonly used functions [e.g., C standard library (`libc`), math library (`libm`)]

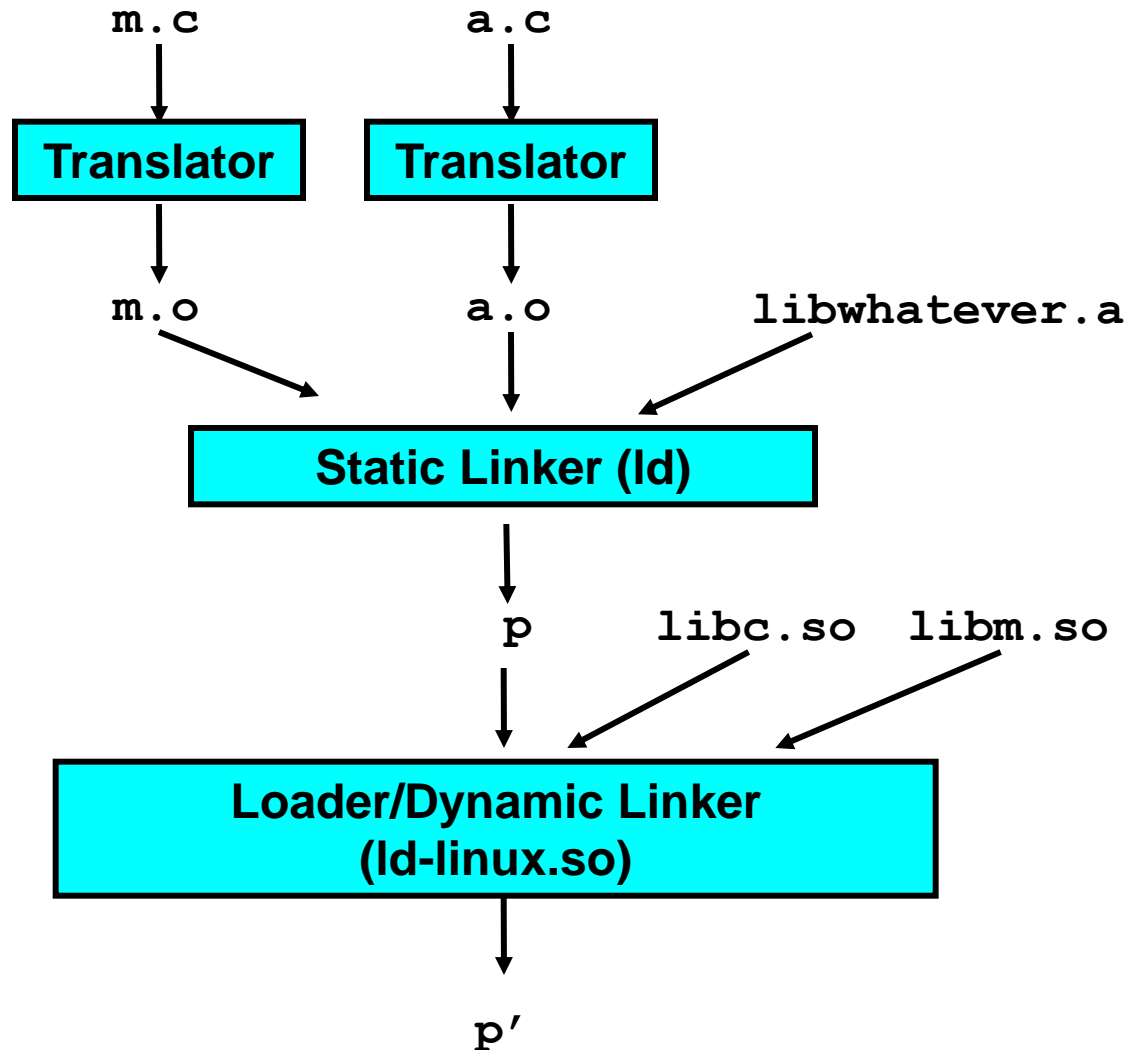
Linker selectively only the `.o` files in the archive that are actually needed by the program.



# Dynamisk länkning



# Blandning av statisk / dynamisk



# Windows och länkning

---

- I MS Windows (XP...) används motsvarande
  - Statisk länkning
    - Kod från bibliotek (.a) bifogas till den exekverbara filen
  - Dynamisk länkning
    - Kod från ett bibliotek (.a) innehållande stub:s bifogas till den exekverbara filen
      - stub: Kod som laddar en rätt DLL (Dynamic Link Library) och överför kodexekveringen till detta funktionen i DLL:n

# Makefiles

---

- C-projekt blir snabbt stora
  - Många källkodsfiler
  - Bibliotek spridda på olika ställen
  - Olika konfigurationer (t.ex. via **#defines**)
- Att manuellt skriva kommandona för att bygga systemet blir invecklat, pga. många långa parametrar, ex:

```
gcc -I/usr/local/include/ -
 L/usr/local/lib -DDEBUG -DWIN32
myfile.c -g -O2 -Wall -o myfile.o -lm
-lcrypt -lext
```

# Makefiles

---

- Istället bygger man en s.k. Makefile som innehåller regler för hur ett programpaket skall byggas
- Baserar sig på
  - definitioner, tex.
    - kataloger där filer finns
    - flaggor / parametrar
  - beroendeförhållanden, tex.
    - att en viss objektfil byggs från vissa källkodsfiler
    - vissa system byggs utgående från vissa objektfiler
  - regler för hur man bygger komponenter
    - hur en kompilator / linker skall aktiveras

# Makefile - exempel

---

```
#-----Macros-----

for cs machines
#BASEDIR = /usr/project/courses/cps008/lib
for acpub machines
BASEDIR = /afs/acpub.duke.edu/users8/ola/courses/lib

TLIB = $(BASEDIR)/libtapestry.a
INCLUDES = -I. -I$(BASEDIR)

set up compiler and options
CXX = g++
CXXFLAGS = -g $(INCLUDES)
```

# Makefile - exempel

---

```
#-----Suffix Rules-----
set up C++ suffixes and relationship between
 .cc and .o files

.SUFFIXES: .cc

.cc.o:
-> $(CXX) $(CXXFLAGS) -c $<

.cc :
-> $(CXX) $(CXXFLAGS) $< -o $@ -lm $(TLIB) -
lg++
 NOTE!! → correspond to a TAB
```

# Makefile - exempel

---

```
#-----File Dependencies-----
SRC = application.cc menu.cc menuitem.cc pixmap.cc
 usepix.cc readcommand.cc quitcommand.cc filelister.cc
 templateapp.cc displaycommand.cc
OBJ = $(addsuffix .o, $(basename $(SRC)))
usepix: $(OBJ)
 -> $(CXX) $(CXXFLAGS) -o $@ $(OBJ) -lm $(TLIB) -
 lg++

#-----Other stuff-----
depend:
 -> makedepend $(CXXFLAGS) -Y $(SRC)
clean:
 -> rm -f $(OBJ)
```



# Makefiles - delar

---

- Makrodefinitioner – text-substitueringar
  - BASEDIR = /export/home/appdev/
  - Används via \$BASEDIR
- Specialkommandon
  - OBJ = \$(addsuffix .o, \$(basename \$(SRC)))
  - Byter ut alla filsuffix mot .o

# Makefile - delar

---

- Beroendeförhållanden

```
usepix: $(OBJ)
```

```
-> $(CXX) $(CXXFLAGS) -o $@ $(OBJ) -lm $(TLIB) -lg++
```

- Obs ”->” svarar för tab-tecknet
- Dvs. usepix är beroend av alla objektfiler, och raden under berättar hur man skall bygga ”usepix”
- Från kommandoraden aktiverar man genom att skriva

```
$ make usepix
```

# Speciella beroenden

---

- Första beroendeförhållande aktiveras som "default" (dvs bara "make" på kommandoraden)
- Pseudo-targets: Egentlig målfil finns ej, men kan användas t.ex. för att bygga andra mål  
`testsuite: test1 test2`
- Mål utan beroenden kan användas för att exekvera vissa kommandon  
`clean:`  
`-> rm -f $(OBJ)`