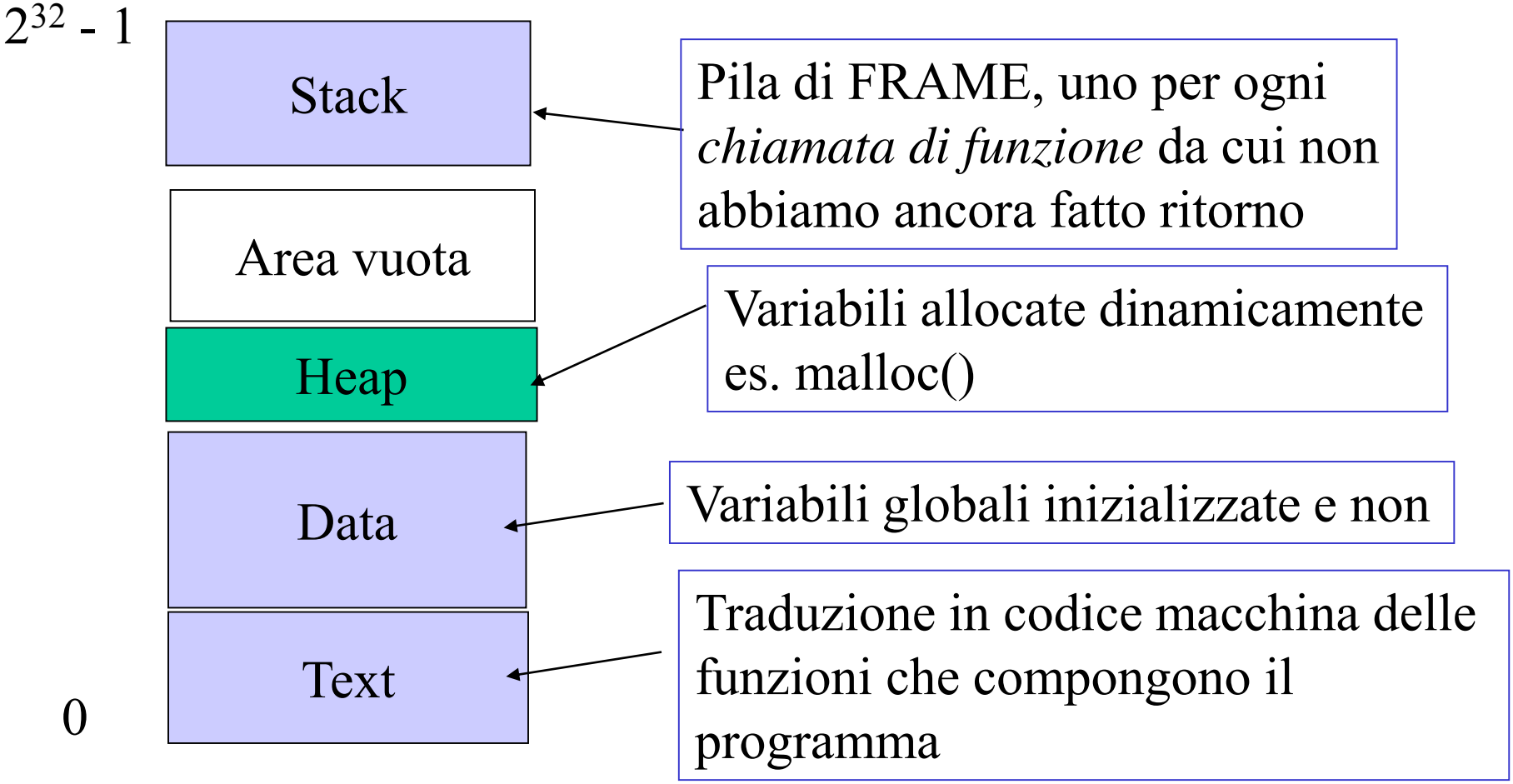
Preprocessing, compilazione ed esecuzione

Utilizzando strumenti GNU...

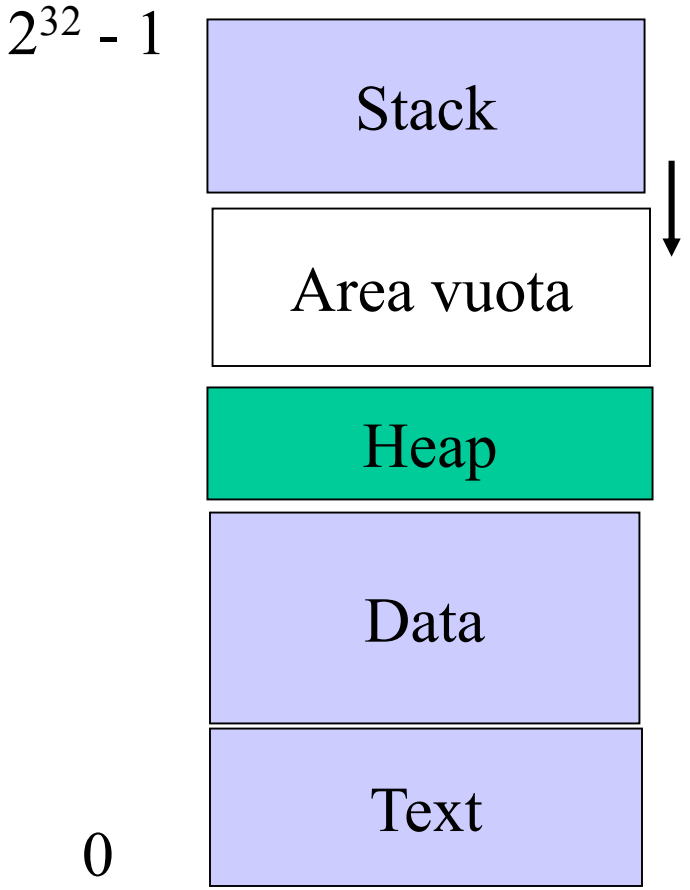
Spazio di indirizzamento

- Come vede la memoria un programma C in esecuzione



Spazio di indirizzamento (2)

- Lo stack

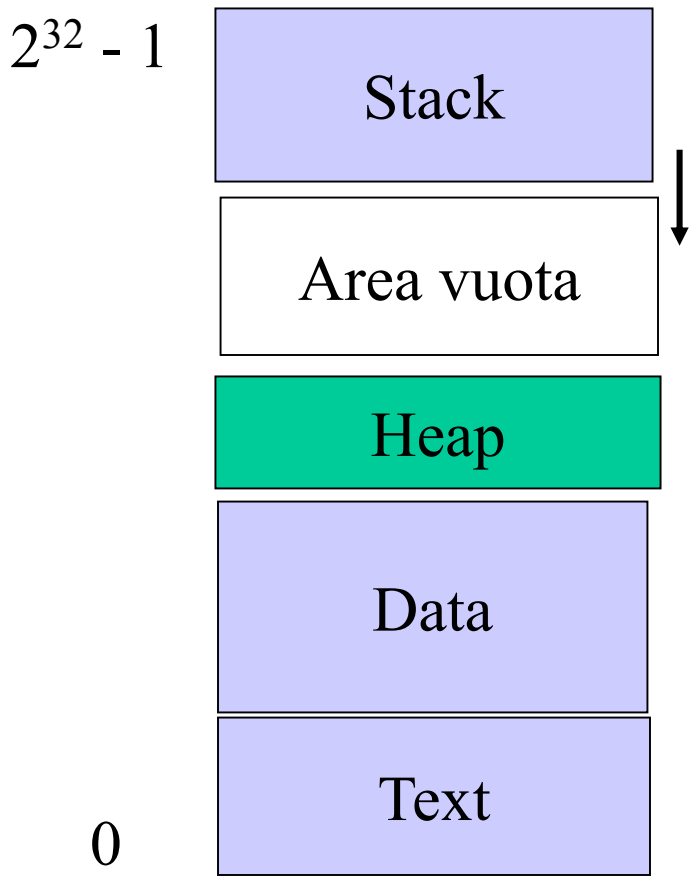


Direzione di
crescita dello stack

Contenuti tipici di un FRAME :
- variabili locali della funzione
- indirizzo di ritorno (indirizzo dell'istruzione successiva a quella che ha effettuato la chiamata alla funzione)
- copia del valore parametri attuali

Spazio di indirizzamento (3)

- Lo stack

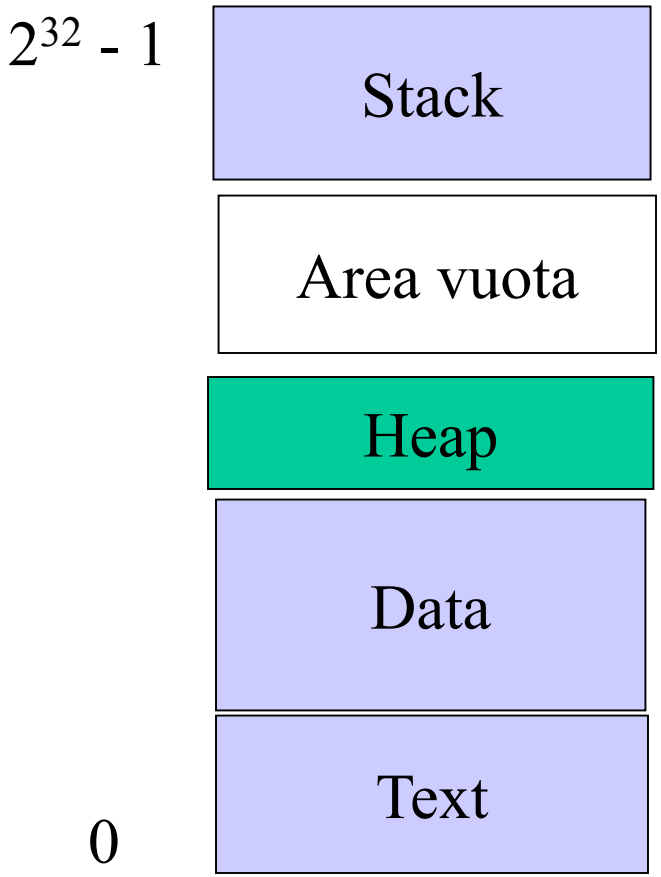


All'inizio dell'esecuzione lo Stack contiene solo il FRAME per la funzione `main`

Successivamente :

* ogni volta che viene chiamata una nuova funzione viene inserito un nuovo frame nello stack
* ogni volta che una funzione termina (`es. return 0`) viene eliminato il frame in cima allo stack e l'esecuzione viene continuata a partire dall'*indirizzo di ritorno*

Spazio di indirizzamento (4)



Direzione di
crescita dello *heap*

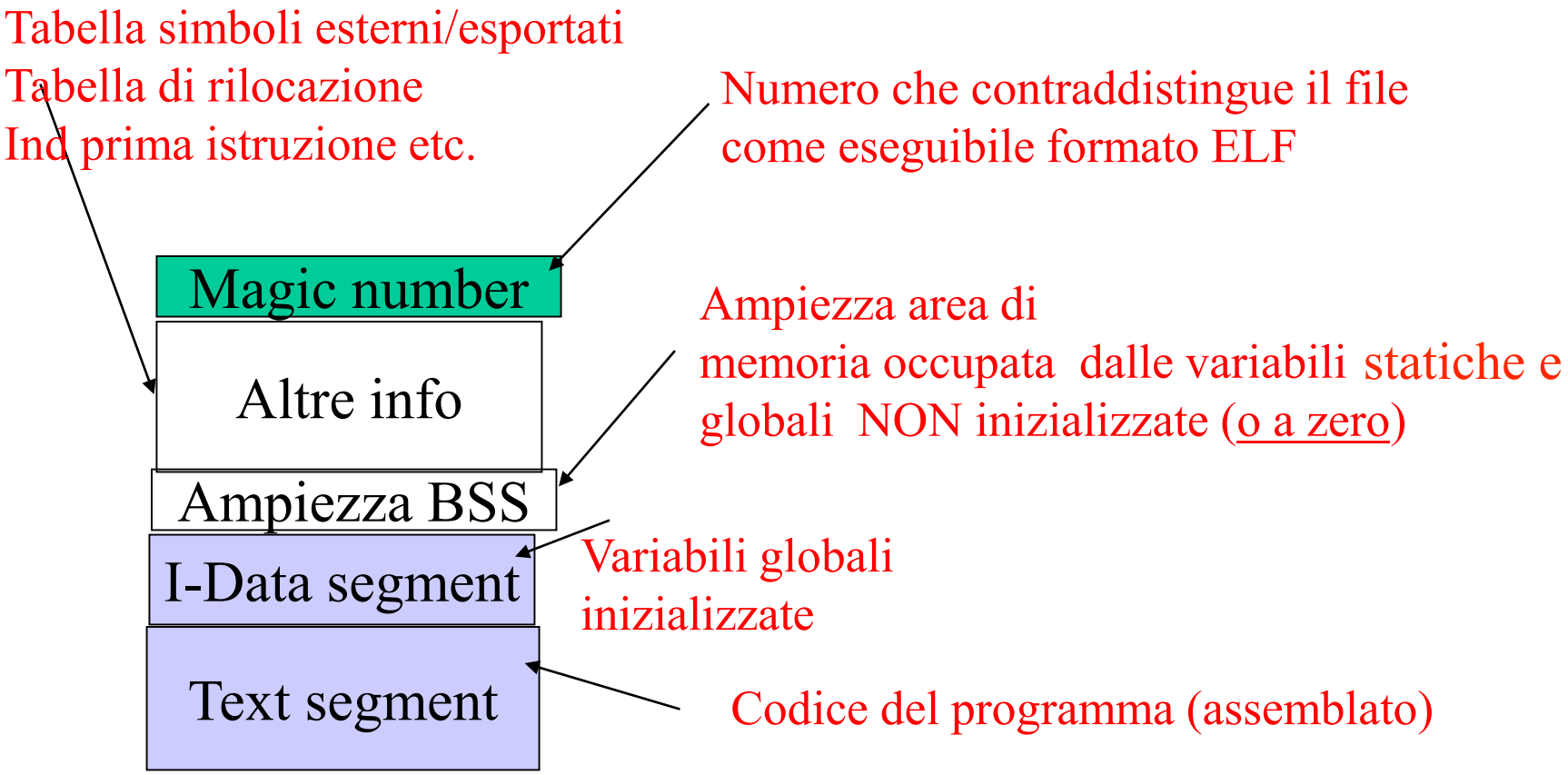
malloc() calloc() realloc() fanno crescere lo heap verso l'alto
la memoria viene deallocata solamente con chiamate alla funzione free() . Se la free() non viene invocata la memoria rimane occupata (*memory leak*)

Formato del file eseguibile

- La compilazione produce un file eseguibile
- Il formato di un eseguibile dipende dal sistema operativo
- In Linux un eseguibile ha il formato ELF (*Executable and Linking Format*)
  - eseguibili e moduli oggetto hanno lo stesso formato
  - assembler + tabelle varie
  - tabelle eliminabili con il comando **`strip`**

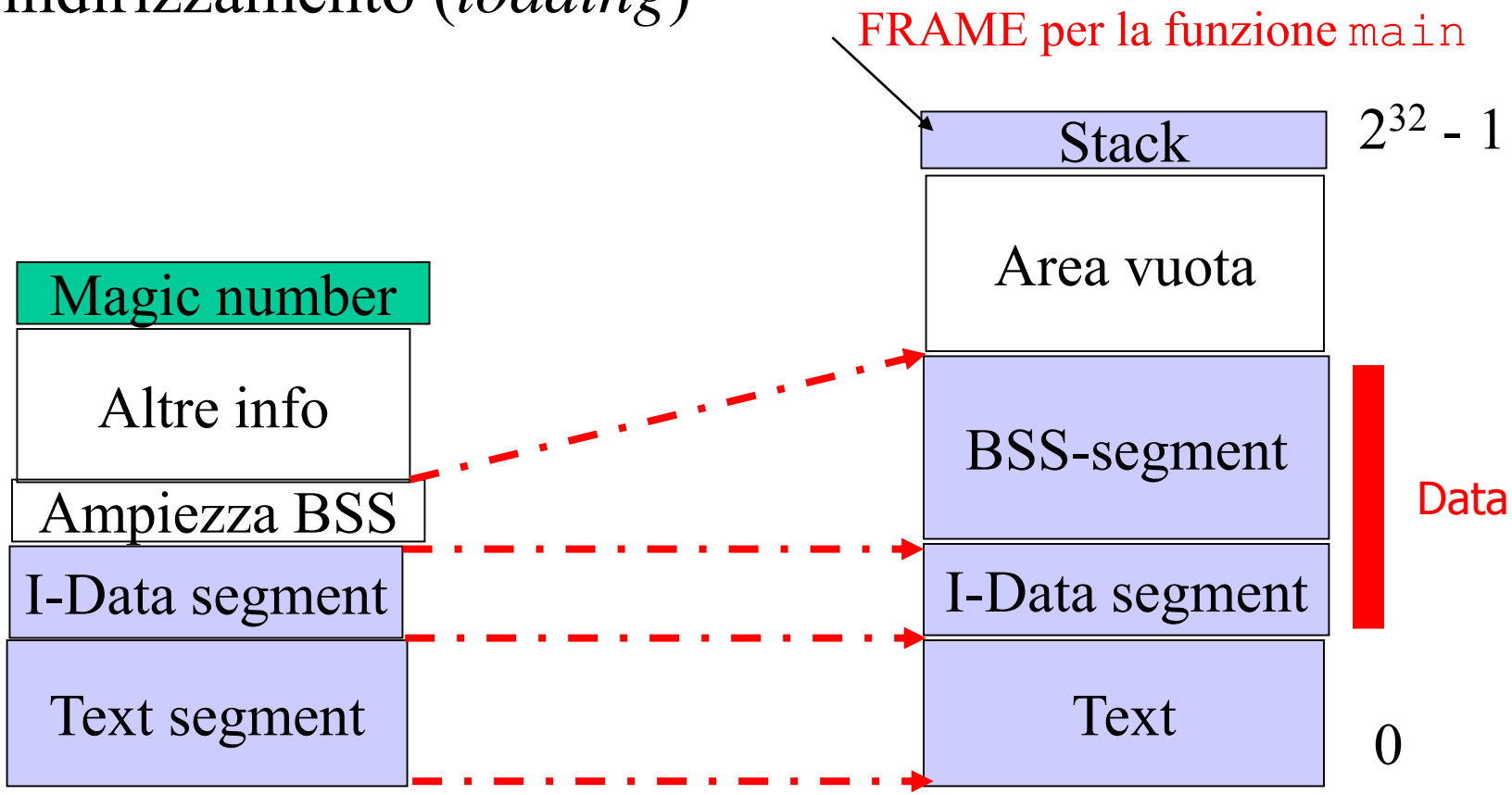
Formato del file eseguibile (2)

- Formato di un eseguibile ELF
  - leggibile con **`readelf, objdump, nm`**



Formato del file eseguibile (3)

– L'eseguibile contiene tutte le informazioni per creare la configurazione iniziale dello spazio di indirizzamento (*loading*)

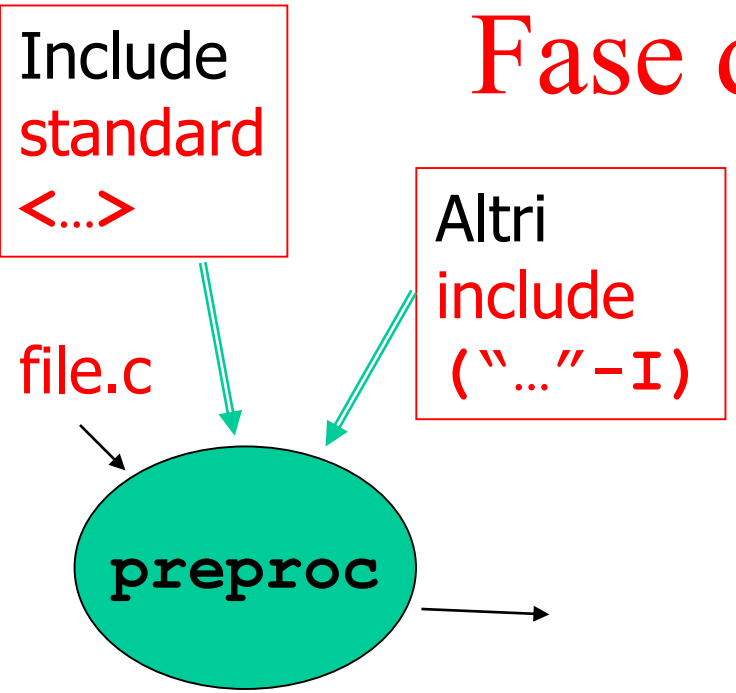


C: dal sorgente all'eseguibile

- Per ottenere l'eseguibile il programma deve essere
  - 1. pre-processato
  - 2. compilato
  - 3. collegato (*linking*)

- Vediamo come funzionano le varie fasi
  - ci riferiremo agli strumenti tipici GNU (**`cpp`**, **`gcc`**, **`ln`**) e al loro utilizzo da shell testuale

Fase di preprocessing




```
cpp prova.c
gcc -E prova.c
```

Preprocessing

- Espansione degli **`#include`**
- Sostituzione della macro (**`#define`**)
- Compilazione condizionale (**`#if #ifdef #endif`**)

Preprocessing: esempio

File **`prova.c`**

direttive per **`cpp`**

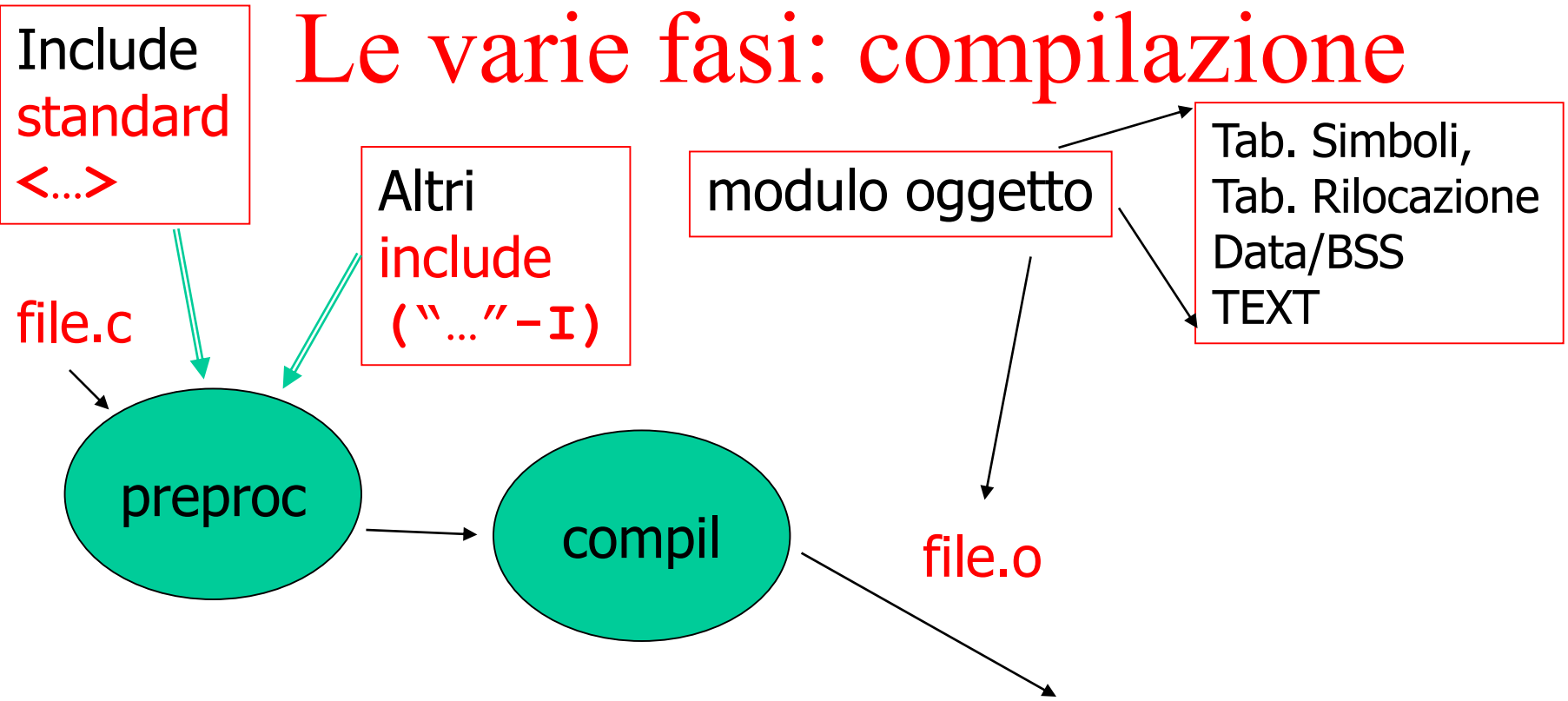
```
#include <stdio.h>
#define N 10
int max = 0;
int main (void){
  int i, tmp;
  printf("Inserisci %d interi positivi\n",N);
  for (i = 0; i < N; i++) {
     scanf("%d", &tmp);
     max = (max > tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

Preprocessing: esempio (2)

Dopo
```
cpp prova.c
gcc -E prova.c
```

```
…… -- copia di stdio.h
# 2 "prova.c" 2
-- qua era la #define
int max = 0;
int main (void){
  int i, tmp;
  printf("Inserisci %d interi positivi\n",10);
  for (i = 0; i < 10; i++) {
     scanf("%d", &tmp);
     max = (max > tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

Le varie fasi: compilazione




```
gcc -c file.c
   (modulo oggetto in file.o)
gcc -S file.c
   (assembler simbolico text e data
     in file.s)
```

File OGGETTO

Un file oggetto contiene le seguenti informazioni:

1. Header: dice dove si trovano le varie sezioni
   opzione -h di objdump
2. Text: il codice
3. Data: data e bss
4. Simboli
5. Rilocazione

Assembling/Linking

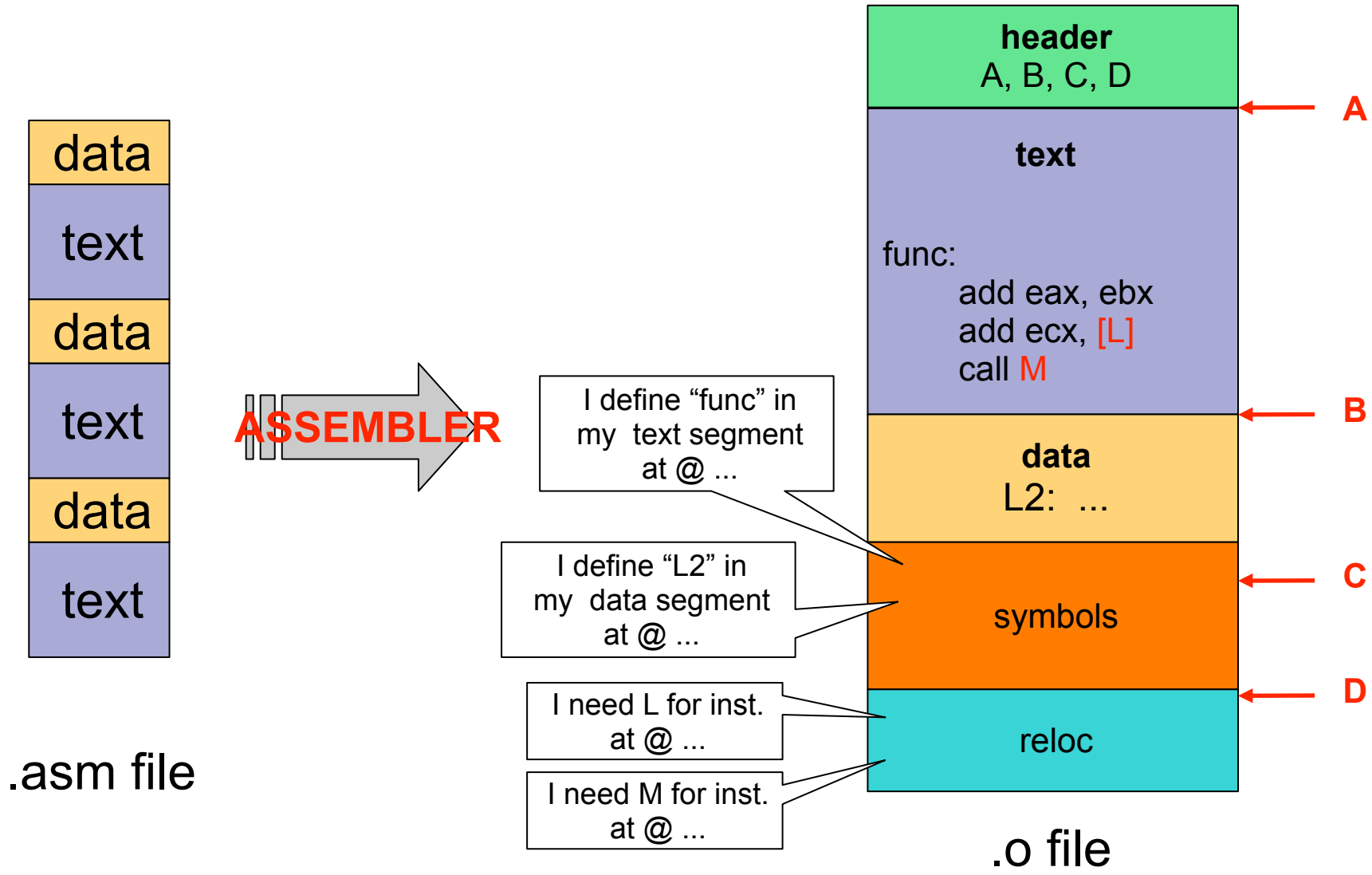


TABELLA DEI SIMBOLI

Memorizza la lista dei “simboli” nel file

Ogni entry memorizza il nome dell’etichetta e il suo OFFSET all’interno di questo file oggetto

TABELLA DI RILOCAZIONE

Memorizza la lista degli “elementi” dei cui indirizzi questo file avrà bisogno (interni o esterni da altri file oggetto e/o librerie)

Compilazione : un esempio

Globali a 0, DATA BSS

```
$ nm prova.o
0000000 T main
0000000 B max
        U printf
        U scanf
```

```
#include <stdio.h>
#define N 10
int max = 0;
int main (void){
  int i, tmp;
  printf("Inserisci %d interi positivi\n",N);
  for (i = 0; i < N; i++) {
     scanf("%d", &tmp);
     max = (max > tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

Compilazione : un esempio

```
#include <stdio.h>
#define N 10
int max = 0;
int main (void){
  int i, tmp;
  printf("Inserisci %d interi positivi\n",N);
  for (i = 0; i < N; i++) {
     scanf("%d", &tmp);
     max = (max > tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

Globali a 0, DATA BSS

```
$ objdump -D prova.o
…
Disassembly of section .bss:
00000000 <max>:
    0:    00 00
……
```

Compilazione : un esempio (2)

Var locali, STACK

tradotte in istruzioni che lavorano sullo stack (TEXT) (le prime del main)

$ objdump -d prova.o

```
#include <stdio.h>
#define N 10
int max = 0;
int main (void){
  int i, tmp;
  printf("Inserisci %d interi positivi\n",N);
  for (i = 0; i < N; i++) {
     scanf("%d", &tmp);
     max = (max > tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

Compilazione : un esempio (3)

codice, TEXT

assembler + 2 call a **`printf()`** e 1 **`scanf()`**

`$ objdump -d prova.o`

```
#include <stdio.h>
#define N 10
int max = 0;
int main (void){
  int i, tmp;
  printf("Inserisci %d interi positivi\n",N);
  for (i = 0; i < N; i++) {
     scanf("%d", &tmp);
     max = (max > tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

Compilazione : un esempio (4)

Simboli esportati (g),
SYMBOL TABLE:
max, main

```
$ objdump -t prova.o
$ nm prova.o
```

```
#include <stdio.h>
#define N 10
int max = 0;
int main (void){
  int i, tmp;
  max=0;
  printf("Inserisci %d interi positivi\n",N);
  for (i = 0; i < N; i++) {
     scanf("%d", &tmp);
     max = (max > tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

Compilazione : un esempio (5)

Simboli non-definiti (*UND*)
SYMBOL TABLE
printf, scanf

$ objdump -t prova.o

```
#include <stdio.h>
#define N 10
int max = 0;
int main (void){
  int i, tmp;
  printf("Inserisci %d interi positivi\n",N);
  for (i = 0; i < N; i++) {
     scanf("%d", &tmp);
     max = (max > tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

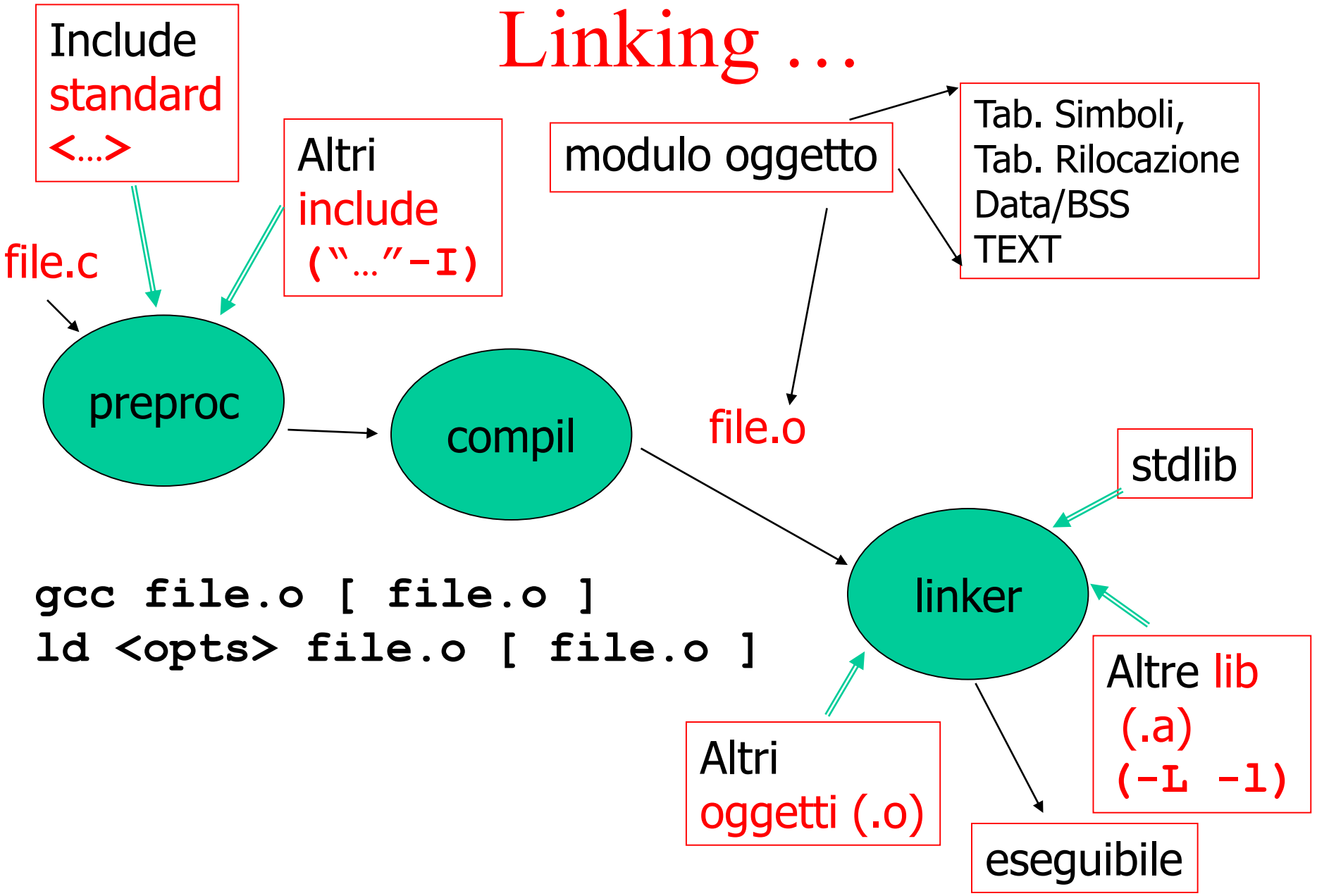
Compilazione : un esempio (6)

Indirizzi da rilocare
RELOCATION RECORDS
max, printf, scanf

$ objdump -r prova.o

```
#include <stdio.h>
#define N 10
int max = 0;
int main (void){
  int i, tmp;
  printf("Inserisci %d interi positivi\n",N);
  for (i = 0; i < N; i++) {
     scanf("%d", &tmp);
     max = (max > tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

Linking …

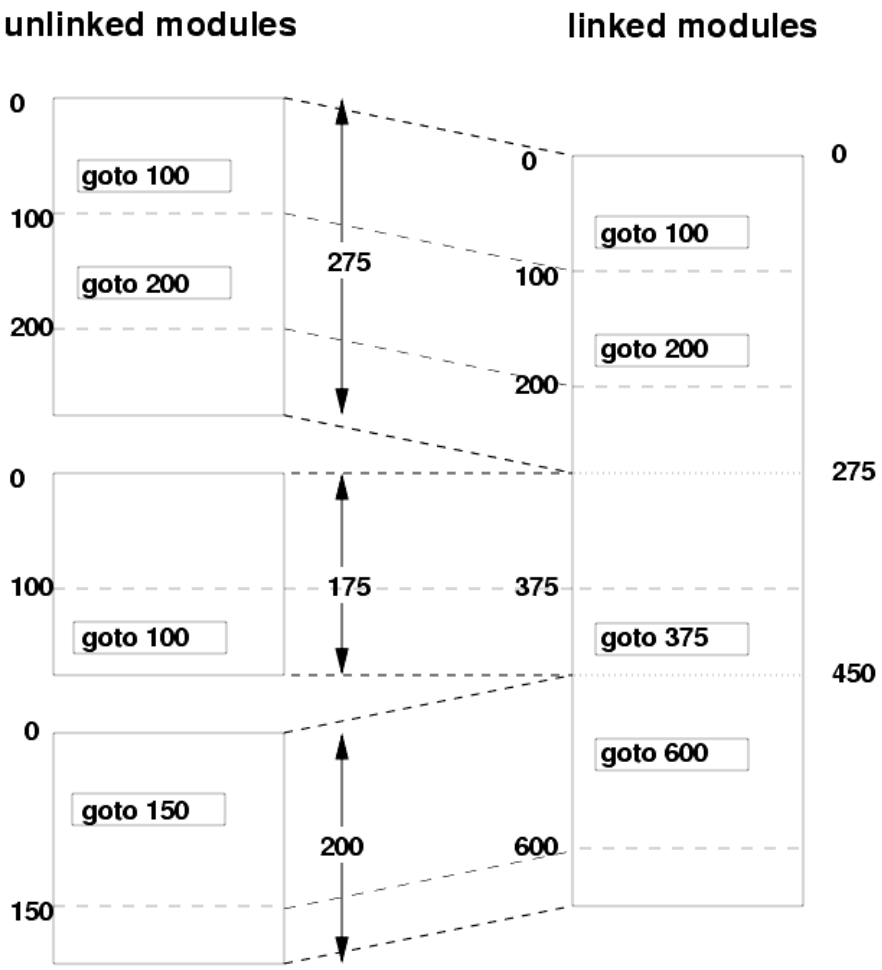



```
gcc file.o [ file.o ]
ld <opts> file.o [ file.o ]
```

Funzioni del Linker 1: Sistemare gli indirizzi

- Indirizzi in un file oggetto sono relativi all'inizio del codice in quel file
- Quando diversi file oggetto sono combinati:
  - Segmenti dello stesso tipo (text, data, ecc.) da diversi file oggetto sono uniti
  - Gli indirizzi devono essere sistemati come conseguenza dell'unione
  - Questo lavoro viene svolto dal linker, utilizzando le informazioni di *rilocazione*

Rilocazione: Esempio



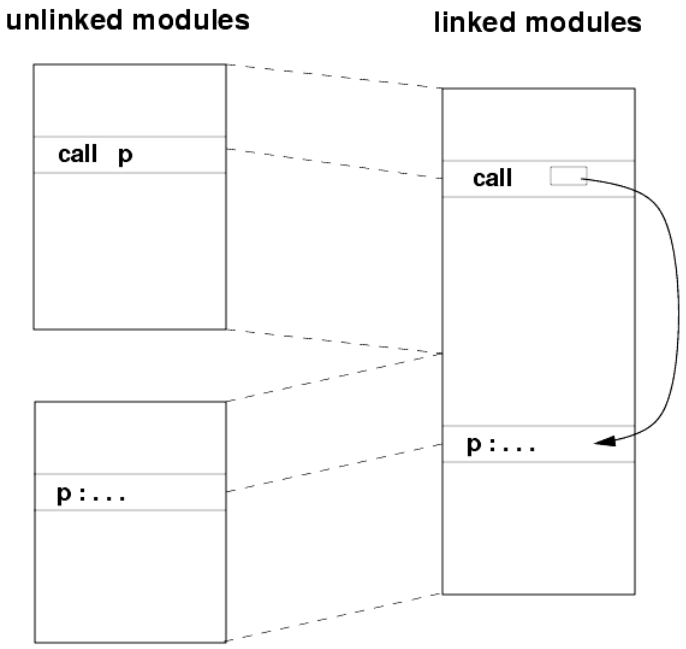
Funzioni del Linker 2: Risoluzione dei simboli

Immaginiamo:

- modulo *B* definisce simbolo p
- modulo *A* riferisce p

Il linker deve:

1. determinare la locazione di p nel modulo oggetto ottenuto dall'unione di *A* e *B*; e
2. modificare i riferimenti a p (in *A* e *B*) per riferire questa locazione



Informazioni per la risoluzione dei simboli

Ogni modulo oggetto contiene una *tabella dei simboli*, che contiene:

- Simboli globali definiti nel modulo
- Simboli globali riferiti ma non definiti nel modulo
- Nomi dei segmenti(*text*, *data*, *ecc*)
  Definiti all'inizio del segmento
- Informazioni opzionali per il debugging

Azioni del Linker

Tipicamente il linker lavora in due passi:

- *Passo 1*:
  - Colleziona le informazioni su tutti i moduli oggetto da linkare
- *Passo 2*:
  - Costruisce l'output, utilizzando tabelle di rilocazione e dei simboli e le info al Passo 1

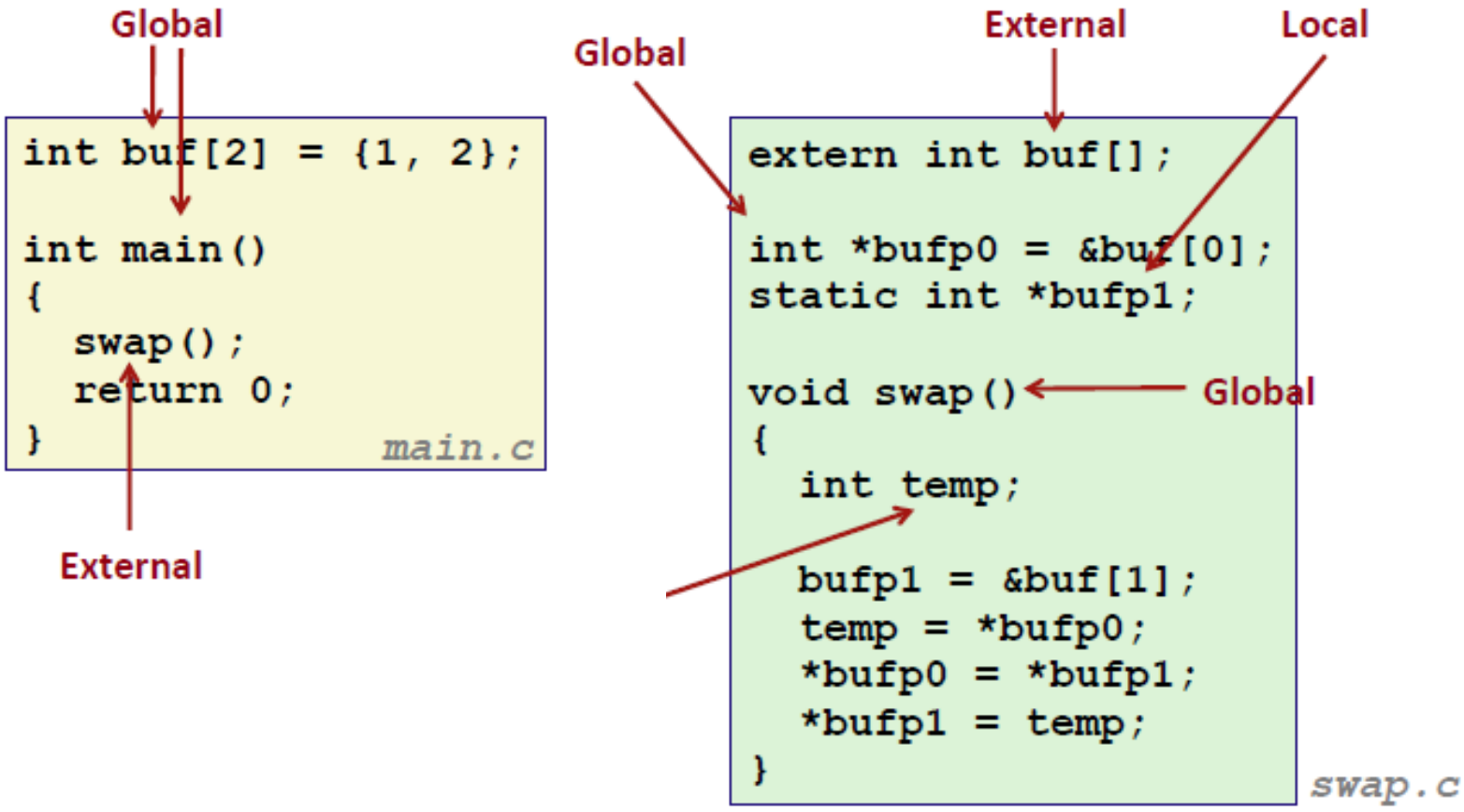
Azioni del Linker: Passo 1

1. Costruisce una tabella di tutti i moduli oggetto, con le relative lunghezze
2. Usando questa tabella, assegna un *indirizzo di caricamento a ogni modulo*
3. Per ogni modulo:
   - Inserisce le sue info nella tabella dei simboli in una *tabella dei simboli globale*
   - Determina l'indirizzo di ogni simbolo definito nel modulo:
     Utilizza il valore del simbolo e l'indirizzo di caricamento del modulo

Azioni del Linker: Passo 2

Copia i moduli oggetto in ordine di indirizzo di caricamento:

1. *Rilocazione degli indirizzi*:
   - trova ogni istruzione che contiene un indirizzo di memoria;
   - a ognuno di questi indirizzi, aggiunge una *costante di rilocazione* che corrisponde all'indirizzo di caricamento di quel modulo
2. *Risoluzione dei simboli esterni*:
   - Per ogni istruzione che riferisce un oggetto esterno, inserisce il nuovo l'indirizzo per quell'oggetto


Global
int buf[2] = {1, 2};
int main()
{
swap();
return 0;
}
main.c
External
Global
External
Local
extern int buf[];
int *bufp0 = &buf[0];
static int *bufp1;
void swap()
Global
{
int temp;
bufp1 = &buf[1];
temp = *bufp0;
*bufp0 = *bufp1;
*bufp1 = temp;
}
swap.c

main.c

```
int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

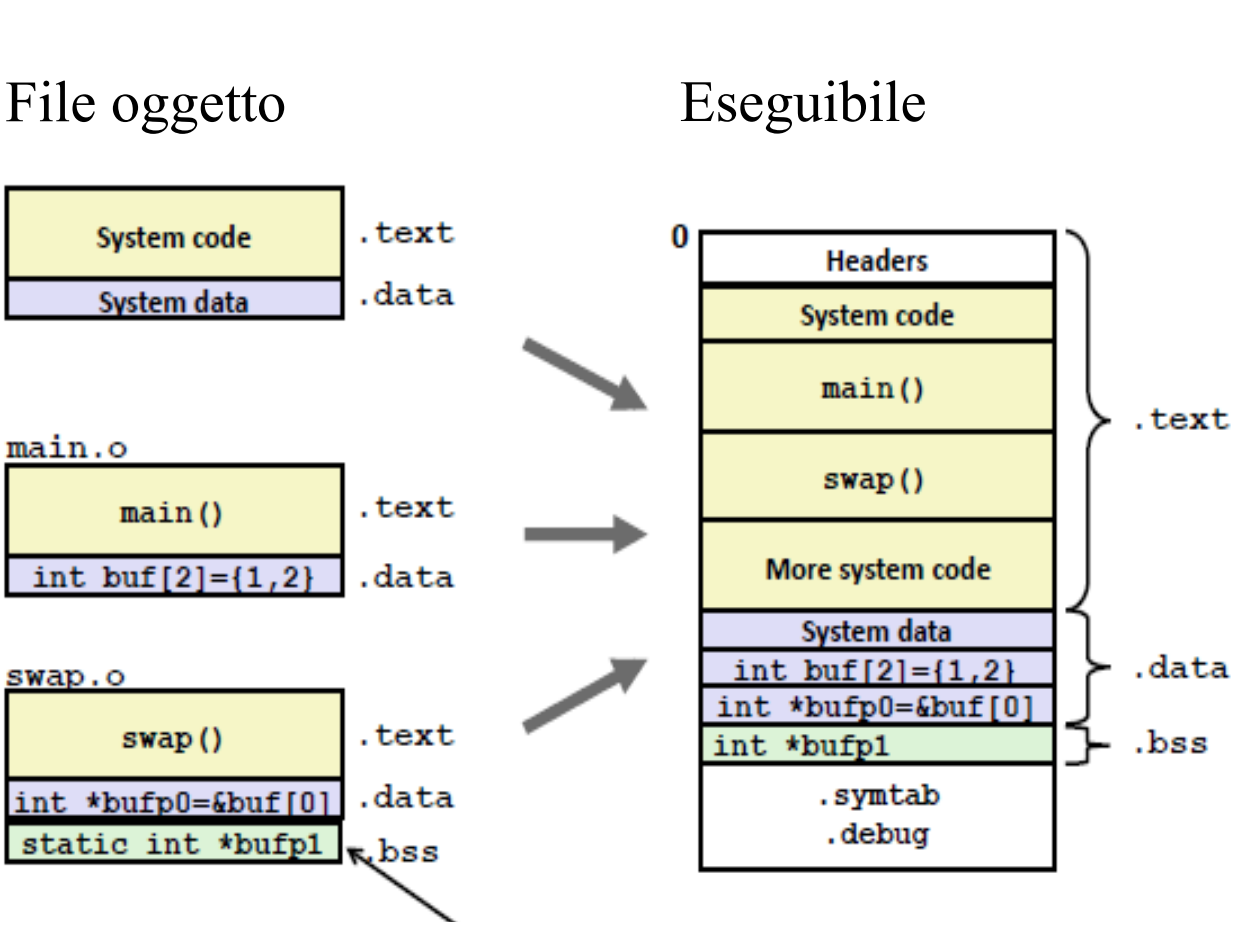
swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```



Linking

Combina diversi file in un unico file eseguibile

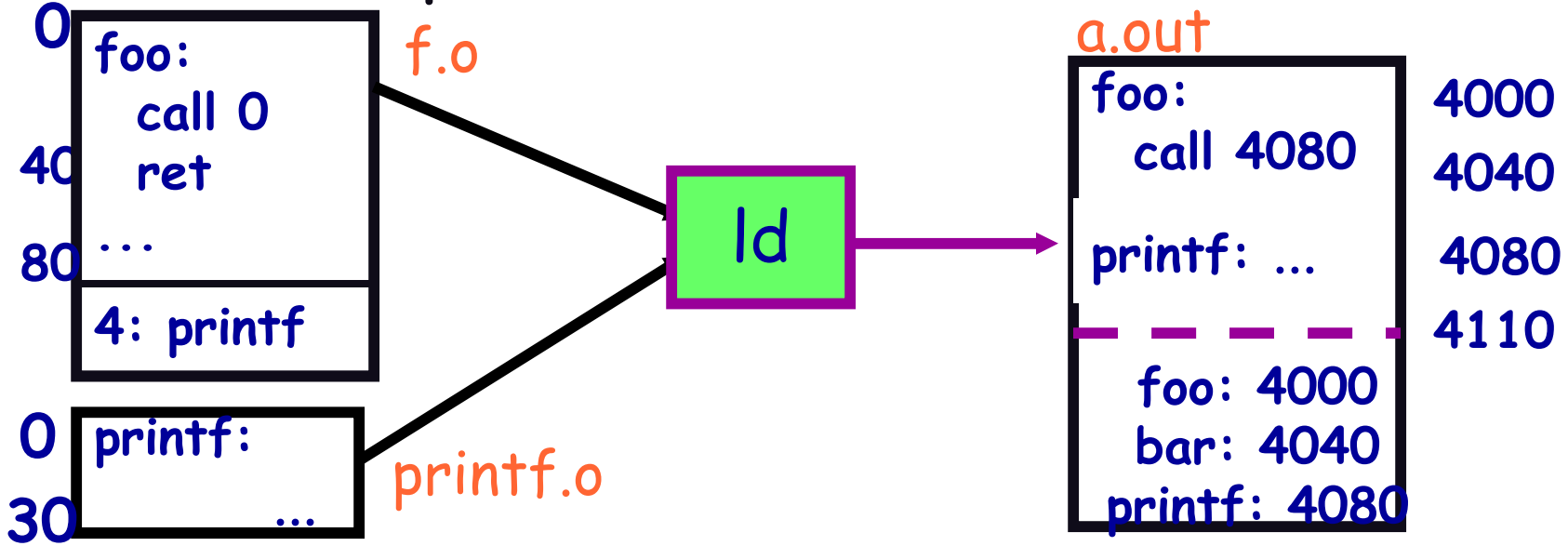
Procede in 3 passi principali:

1. Concatena tutti i TEXT dei file oggetto

2. Concatena tutti i segmenti DATA/BSS dei file oggetto

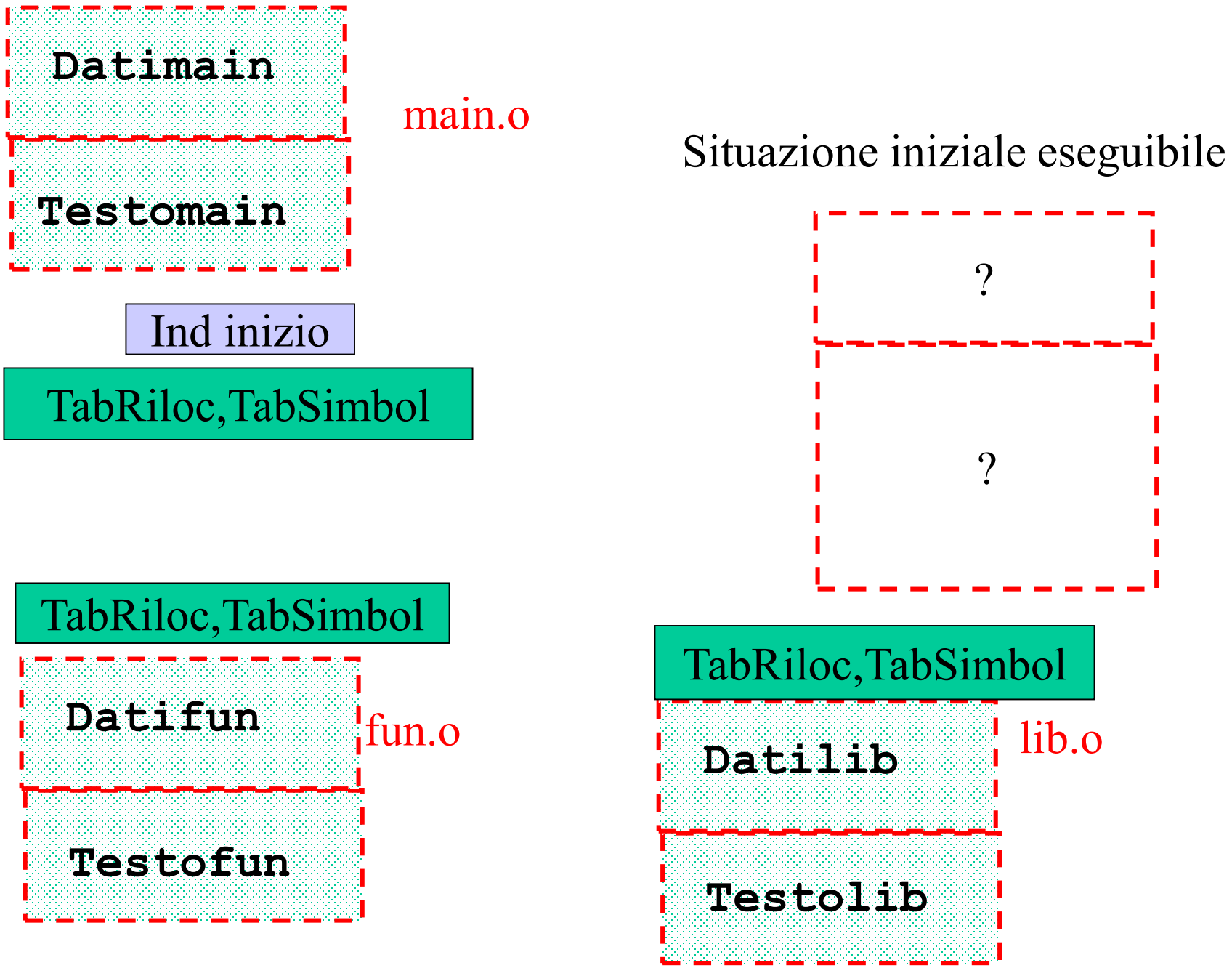
3. Risolve i riferimenti, utilizzando tabella dei simboli e di rilocazione per calcolare gli indirizzi assoluti

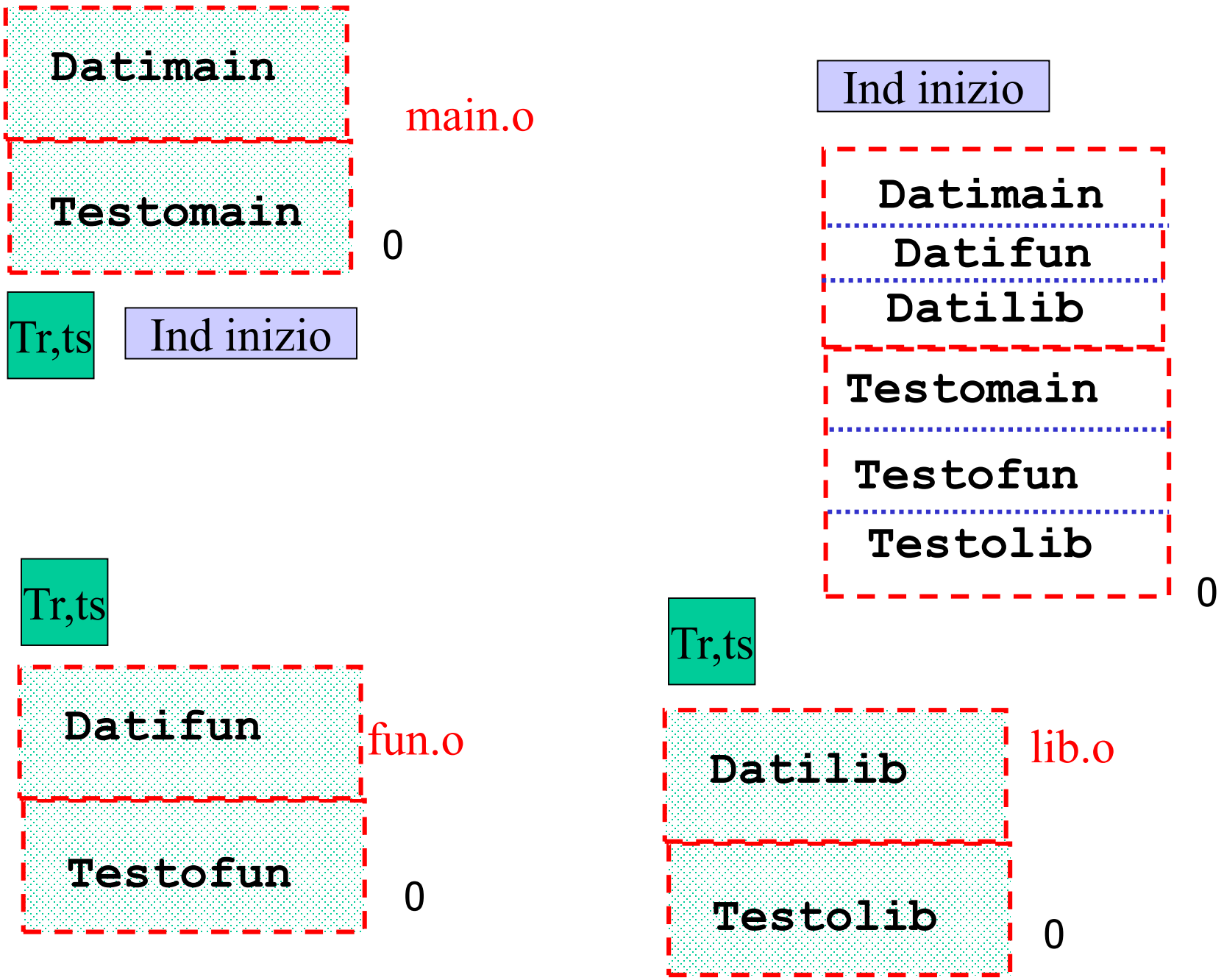
- At link time the linker
  - records all references in the global symbol table
  - after reading all files, each symbol should have exactly one definition and 0 or more uses
  - the linker then enumerates all references and fixes them by inserting their symbol's virtual address into the reference's specified instruction or data location



Linking: esempio statico

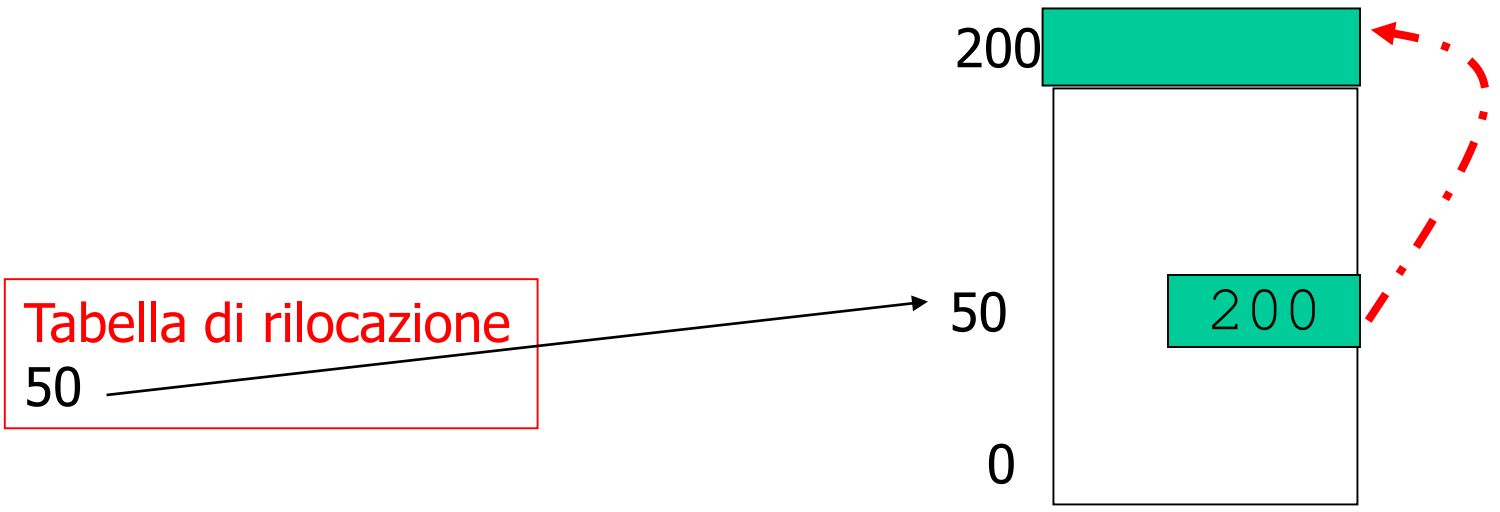
- Si collegano assieme più moduli oggetto
  - **`file.o`** oppure librerie di oggetti **`libfile.a`**
  - ...per creare un file eseguibile
- Ogni modulo oggetto contiene
  - L'assemblato del sorgente **testo e dati** (si assume di partire dall'indirizzo 0)
  - La **tabella di rilocazione**
  - La **tabella dei simboli** (esportati ed undefined)


Datimain
Testomain
main.o
Ind inizio
TabRiloc,TabSimbol
Situazione iniziale eseguibile
?
?
TabRiloc,TabSimbol
Datifun
Testofun
fun.o
TabRiloc,TabSimbol
Datilib
Testolib
lib.o


Datimain
Testomain
main.o
0
Tr,ts
Ind inizio
Ind inizio
Datimain
Datifun
Datilib
Testomain
Testofun
Testolib
0
Tr,ts
Datifun
fun.o
Testofun
0
Tr,ts
Datilib
lib.o
Testolib
0

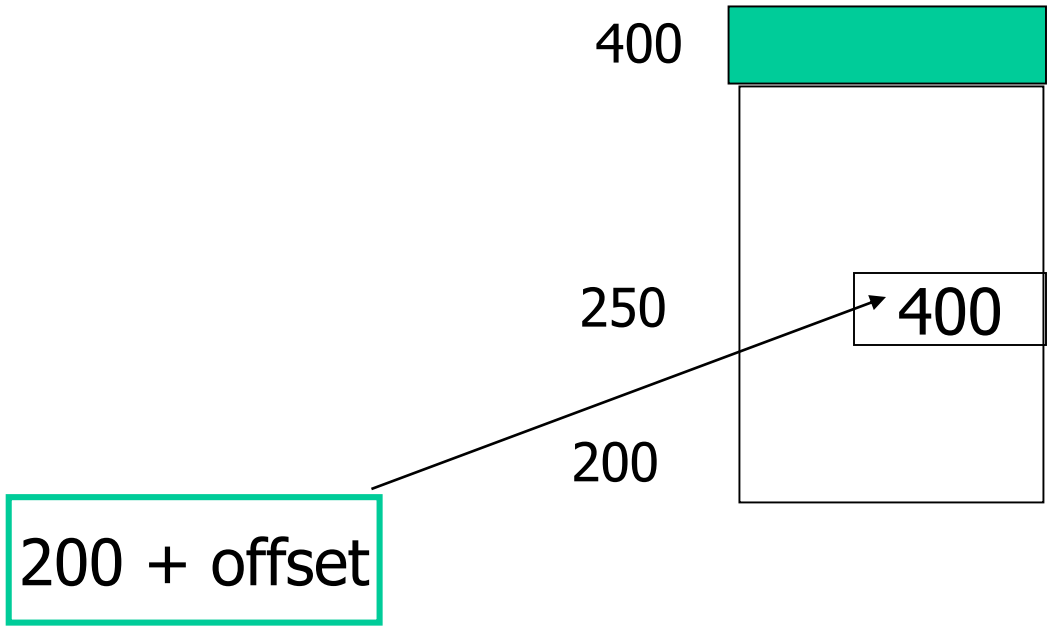
Linking: esempio statico (3)

- Tabella di rilocazione (cont.)
  - il codice pre-compilato è formato da testo e dati binari
  - l'assemblatore assume che l'indirizzo iniziale sia 0



Linking: esempio statico (4)

- Tabella di rilocazione (cont.)
  - es: ad ogni indirizzo rilocabile va aggiunto **`offset = 200,`** l'indirizzo iniziale nell'eseguibile finale



Linking: esempio statico (5)

- Tabella dei simboli
  - identifica i simboli che il compilatore non è riuscito a 'risolvere', cioè quelli di cui non sa ancora il valore perché tale valore dipende dal resto dell'eseguibile finale
  - ci sono due tipi di simboli ...
    - definiti nel file ma usabili altrove (esportati)
      - es: i nomi delle funzioni definite nel file, i nomi delle variabili globali
    - usati nel file ma definiti altrove (esterni)
      - `es:` le funzioni usate nel file ma definite altrove (es. `printf()`)

Linking: esempio statico (6)

- Tabella dei simboli (cont.)
  - per i simboli esportati, la tabella contiene
    - nome, indirizzo locale
  - per i simboli esterni contiene
    - nome
    - indirizzo della/e istruzioni che le riferiscono (come relocation record)

Risoluzione riferimenti

Il linker conosce:
1. La lunghezza di ogni segmento TEXT e DATA
2. Il loro ordine

Quindi il linker può calcolare un indirizzo assoluto per ogni etichetta, assumendo che l'inizio sia all'indirizzo zero

Per ogni etichetta nella tabella di rilocazione, cerca dove è definita (nella tabella dei simboli di qualche file oggetto o libreria)

Se non lo risolve, termina con errore
Se lo trova in più tabelle, termina con errore
Se lo trova in esattamente una tabella, rimpiazza l'etichetta con un indirizzo assoluto

Termina quando non ci sono più riferimenti, ma solo indirizzi

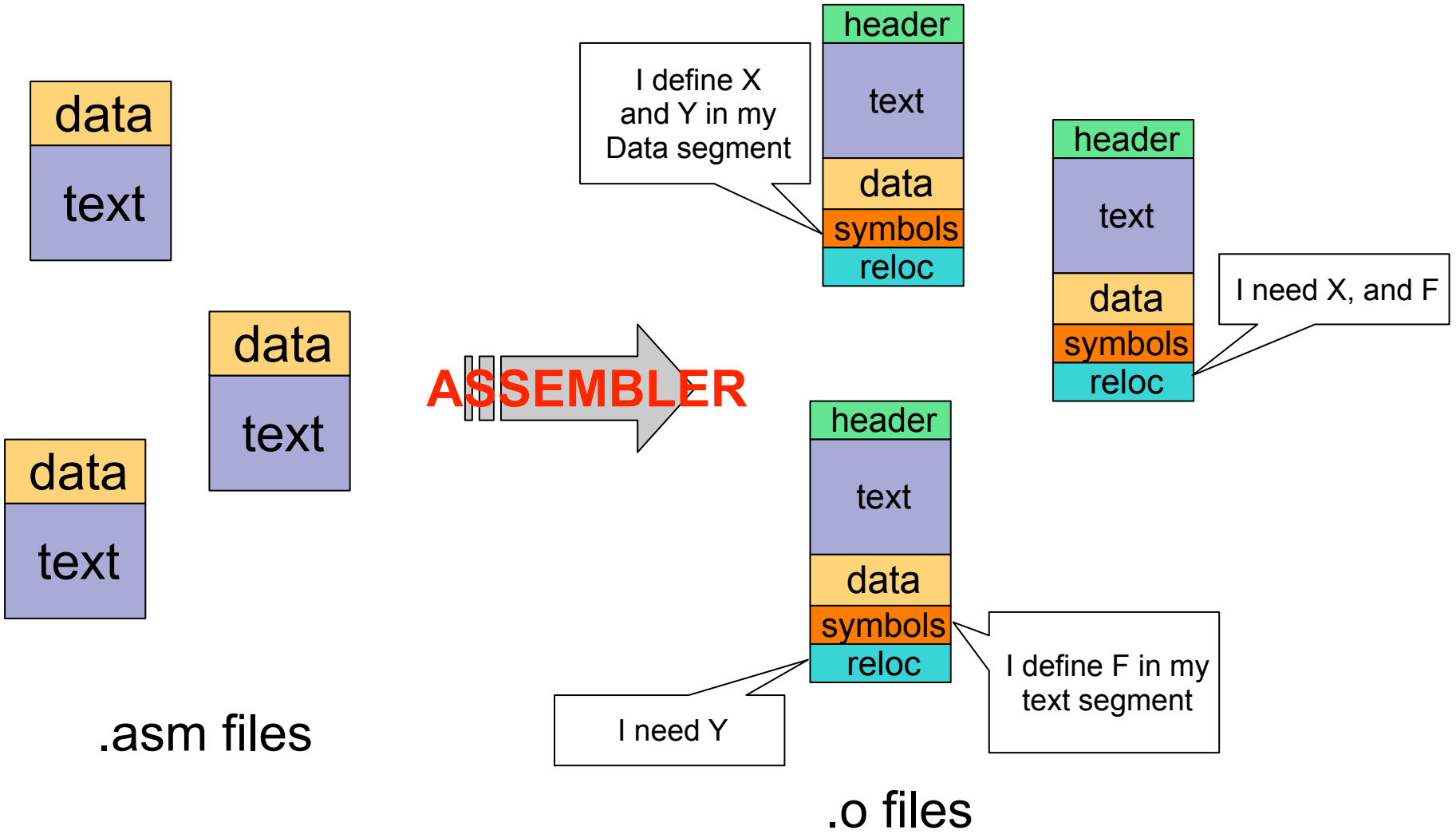
Linking: esempio statico (7)

- Il *linker* si occupa di risolvere i simboli.
  - Analizza tutte le tabelle dei simboli.
  - Per ogni simbolo non risolto (esterno) cerca
    - nelle librerie standard
    - in tutte le altre tabelle dei simboli esportati degli oggetti da collegare (*linkare*) assieme
    - nelle librerie esplicitamente collegate (opzione `-l`)

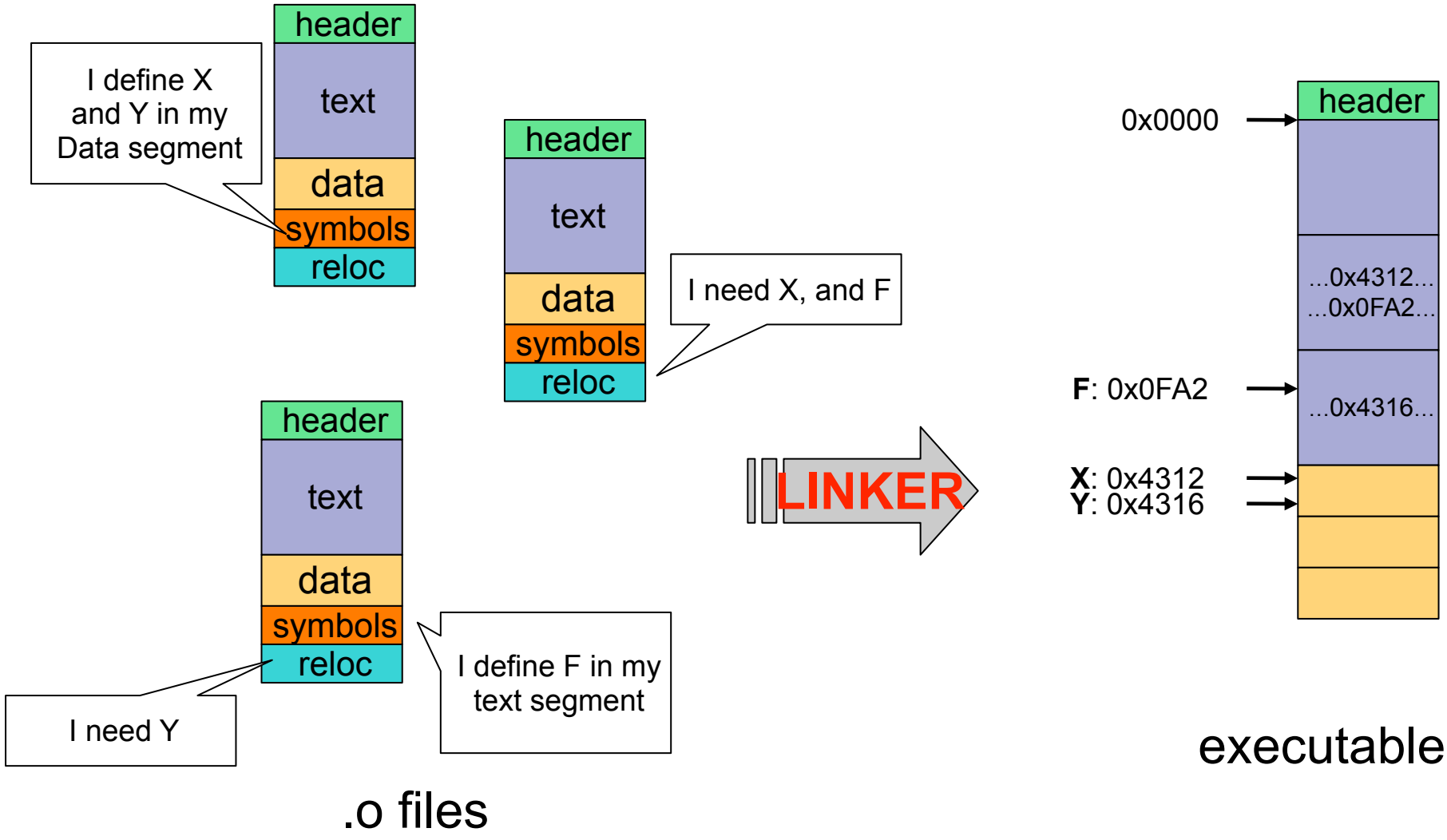
Linking: esempio statico (8)

- Il *linker* si occupa di risolvere i simboli (cont.)
  - Se il linker trova il simbolo esterno
    - ricopia il codice della funzione (linking *statico*) nell'eseguibile
    - usa l'indirizzo del simbolo per generare la CALL giusta o il giusto riferimento ai dati
  - Se non lo trova da errore ...
    - Provate a non linkare le librerie matematiche ...

Assembling/Linking



Assembling/Linking



Caricamento

I programmi sono caricati in un indirizzo di memoria

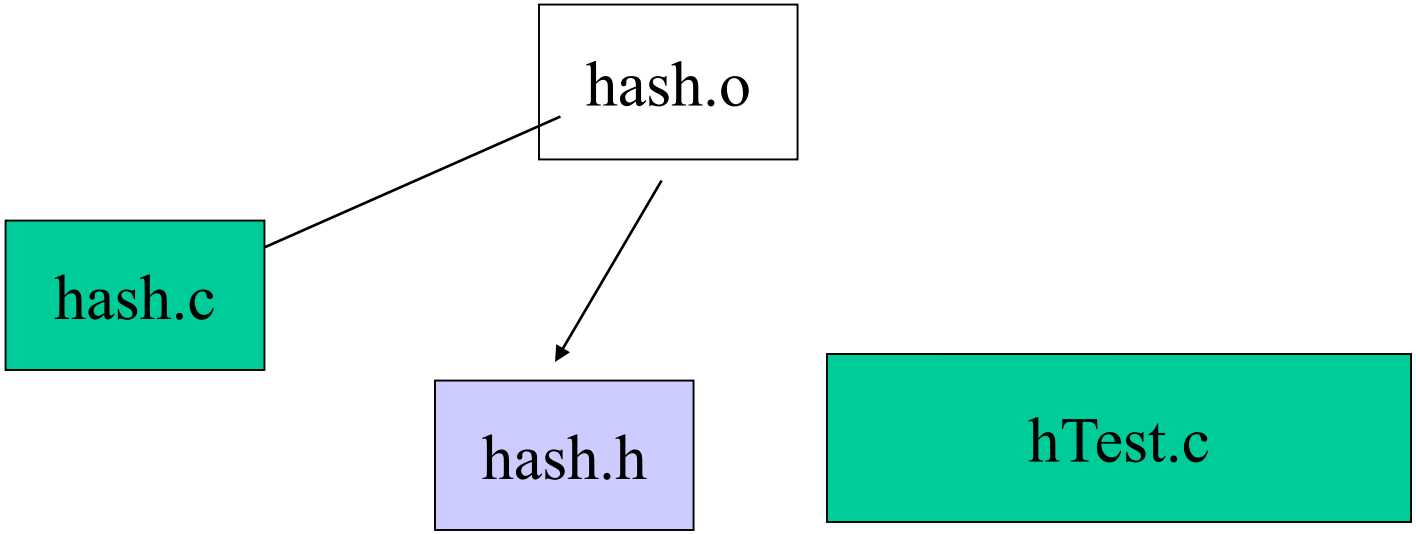
Il caricamento fa le cose seguenti:

1. determina lo spazio necessario dall'header del file eseguibile
2. alloca lo spazio
3. trasferisce il programma nello spazio di indirzzamento
4. azzera le parti non inizializzate (".bss" segment)
5. crea lo stack
6. inizializza informazioni necessarie per l'esecuzione, come argomenti del programma o variabili d'ambiente
7. avvia l'esecuzione del programma

Esempio

Compilare e linkare correttamente un piccolo progetto

Esempio: tabella hash

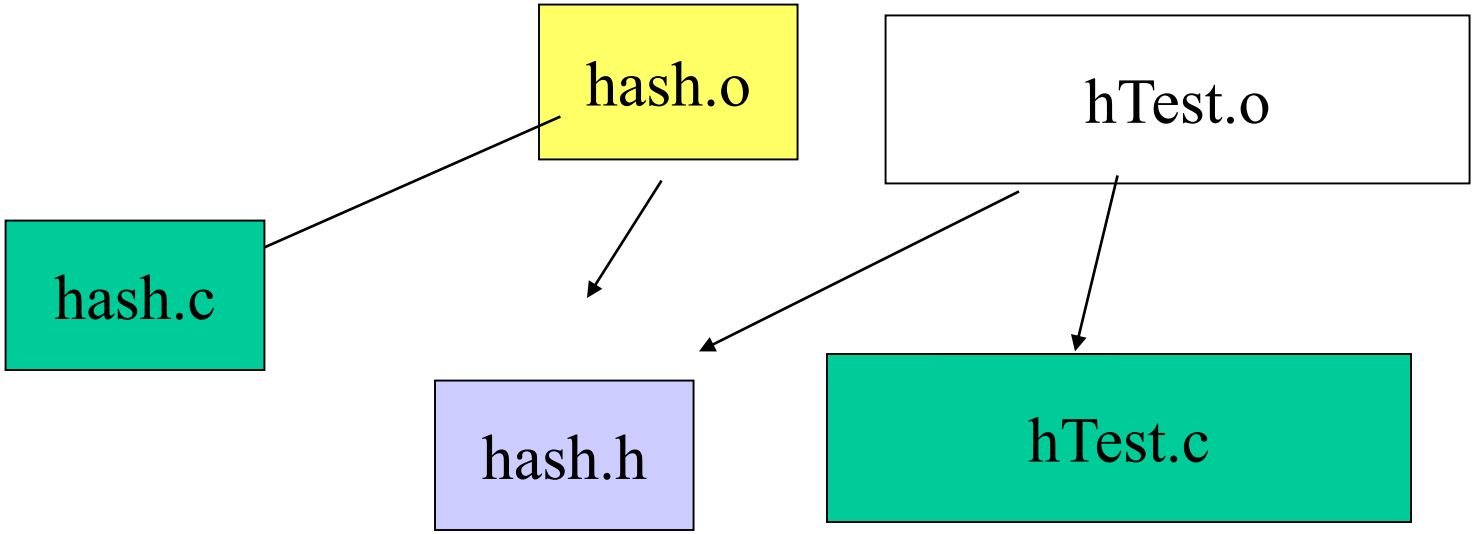


Passo (1):

```
bash:~$ gcc -Wall -pedantic -c hash.c
```

--crea hash.o

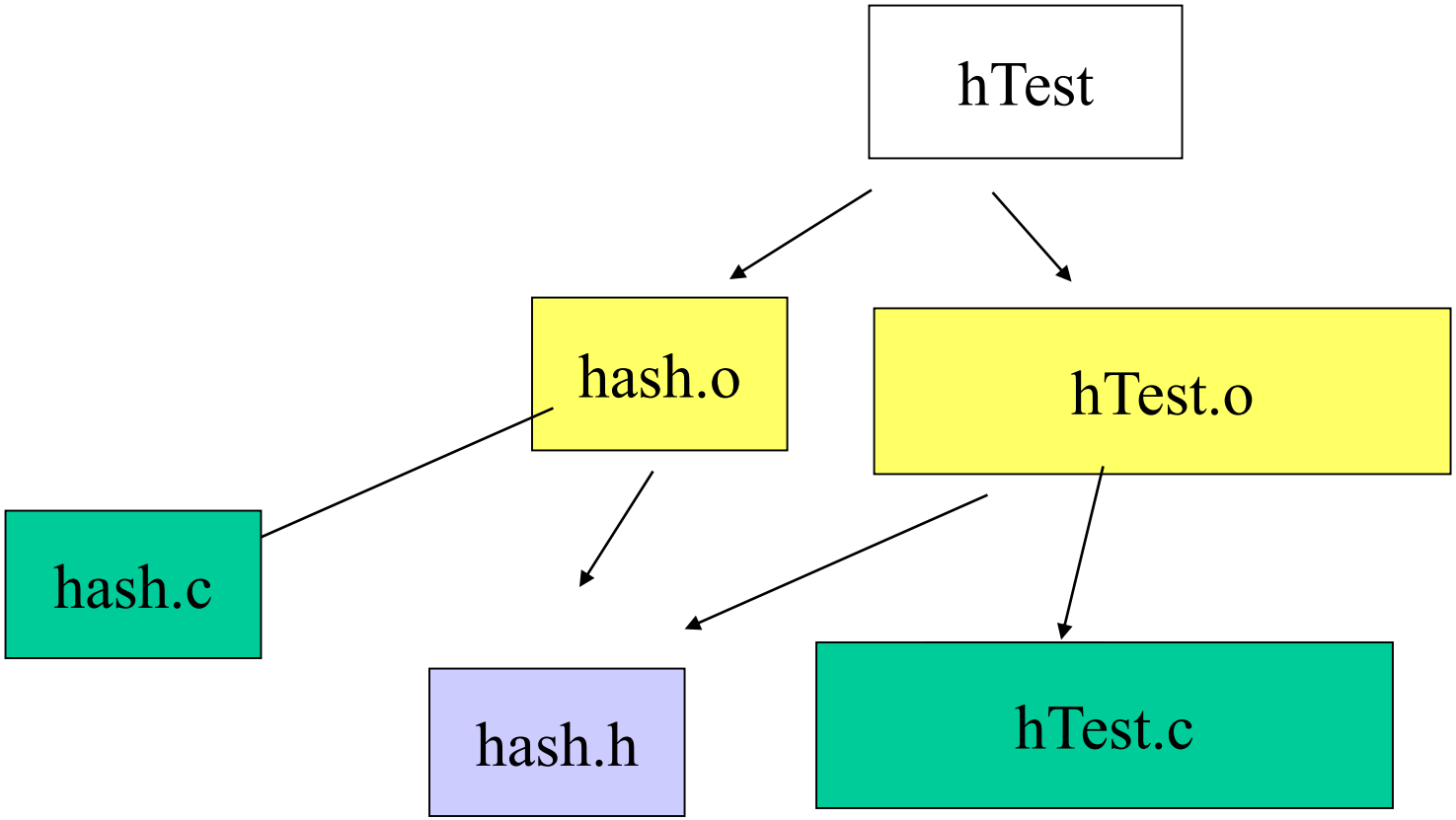
Esempio: hash … (2)



Come costruire l'eseguibile: passo (2):

```
$gcc -Wall -pedantic -c hTest.c
   --crea hTest.o
```

Esempio: hash … (3)



Come costruire l'eseguibile: passo (3):

```
$gcc hash.o hTest.o -o hTest
```

--crea l'eseguibile 'hTest'

Esempio: hash … (4)

```
$gcc -Wall -pedantic -c hash.c        --(1)
$gcc -Wall -pedantic -c hTest.c       --(2)
$gcc hash.o hTest.o -o hTest          --(3)
```

- se modifico **`hash.c`** devo rieseguire (1) e (3)
- se modifico **`hash.h`** devo rifare tutto

- NOTA: per ricreare sempre tutti i moduli oggetto da 0 e rilinkare basta invece

```
$gcc -Wall -pedantic hash.c hTest.c -o hTest
```

Esempio: hash … (5)

```
$ gcc -M hash.c
  --fa vedere le dipendenze da tutti i file
 anche dagli header standard delle librerie
  hash.o : hash.c /usr/include/stdio.h \
           /usr/include/sys/types.h \
           … … … … …
$ gcc -MM hash.c
  hash.o : hash.c hash.h
$
```

- perche' questo strano formato ?
  - per usarlo con il make ….