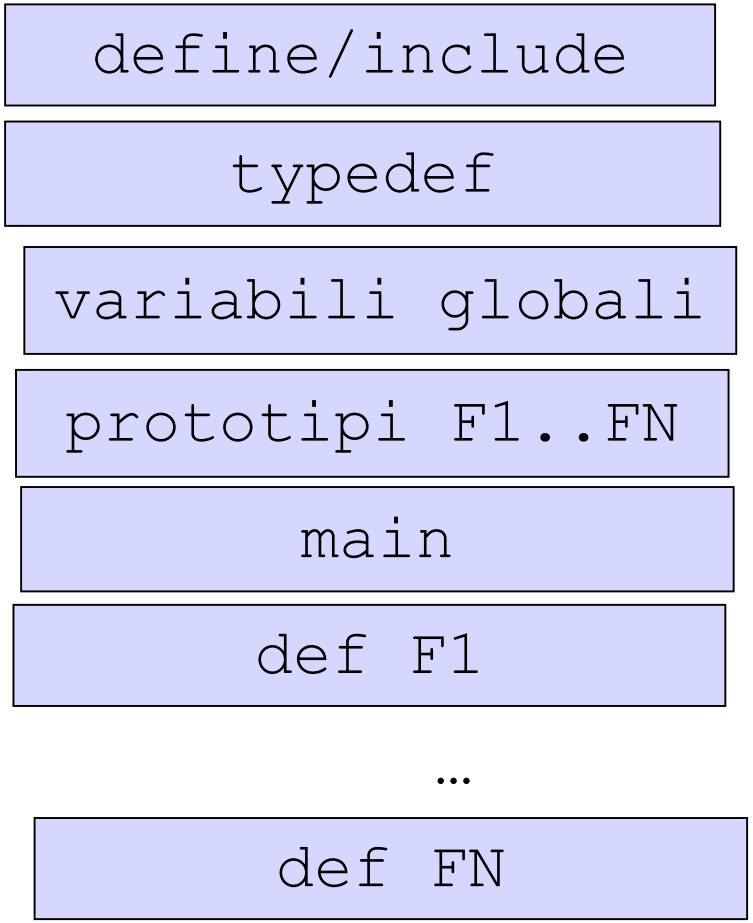
Compilazione separata

Compilazione separata

- Finora abbiamo trattato solo programmi C contenuti in un unico file



Struttura tipica di un sorgente C

Compilazione separata (2)

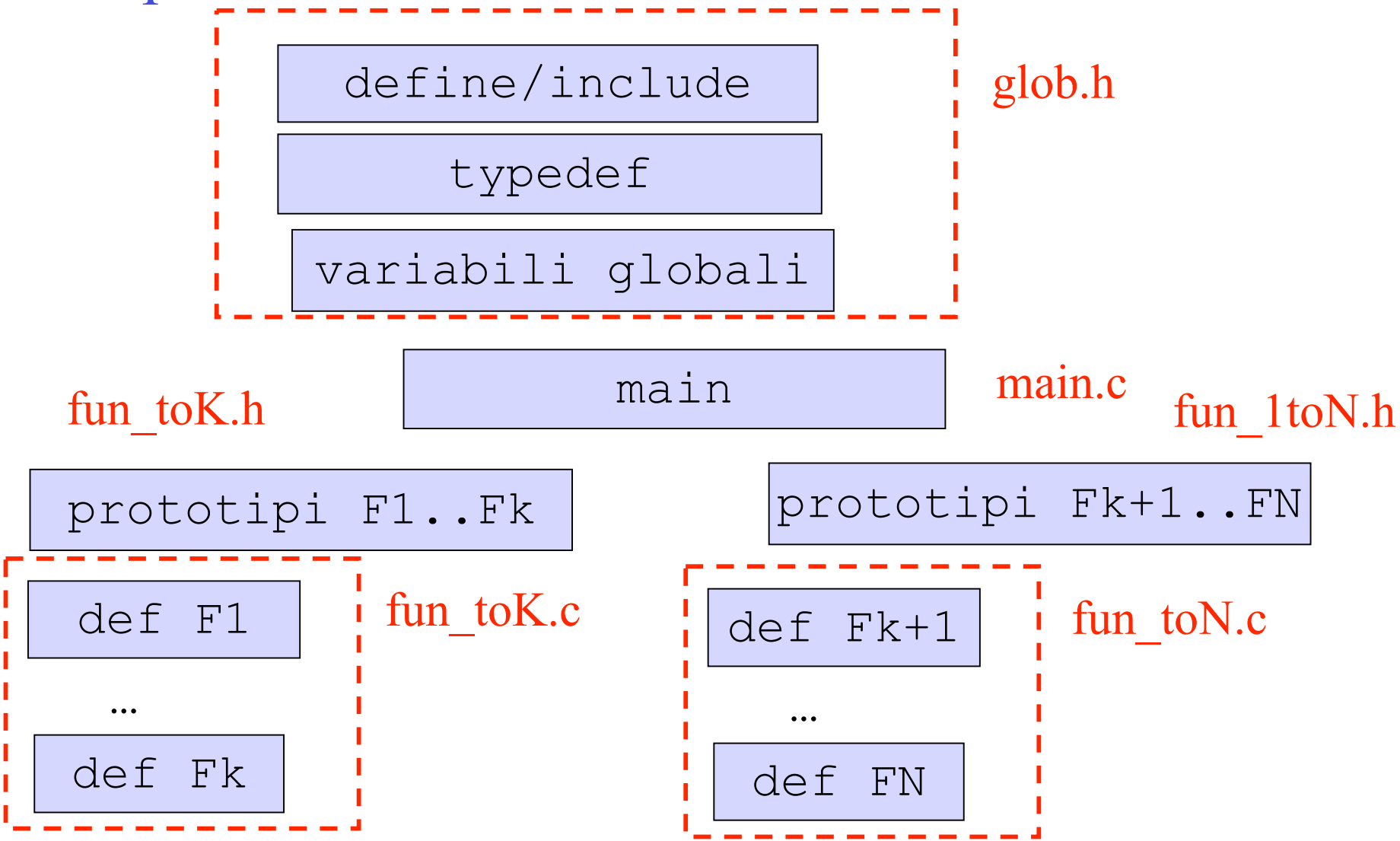
- Scopi di suddividere il codice sorgente C su più file
  - per raggruppare un insieme di funzioni congruenti (e solo quelle) in un unico file
  - per incapsulare un insieme di funzioni esportando solo quelle che vogliamo fare utilizzare agli altri (pubbliche)
  - per compilare una volta per tutte un insieme di funzioni e metterlo a disposizione di altri in una libreria
    - es. le librerie standard viste finora ….

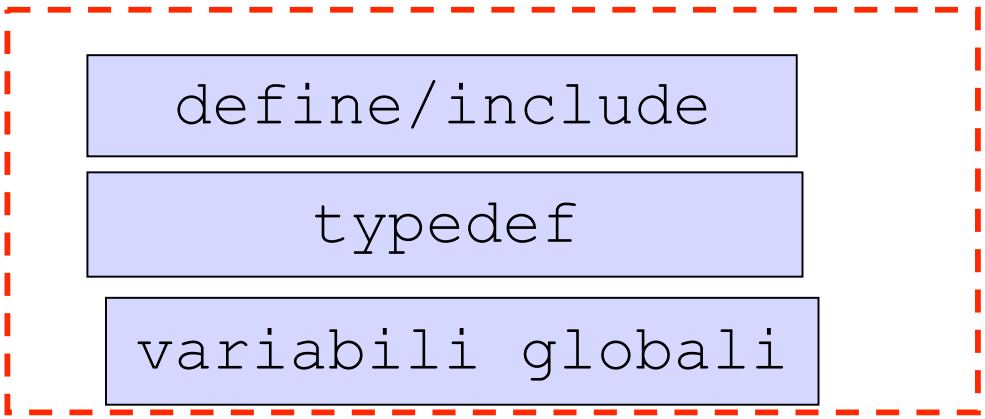
Compilazione separata (2.1)

- Problemi da risolvere
  - (1) come si possono utilizzare funzioni definite in altri file ?
    - Vogliamo che il compilatore continui a fare i controlli di tipo etc …
  - (2) come posso compilare una volta per tutte le funzioni definite in un file separato ?
    - non voglio doverle ricompilare ogni volta che le uso in un altro file (come avviene per printf(), scanf())
    - voglio che il codice assembler risultante possa essere facilmente 'specializzato' per generare l'eseguibile finale

Compilazione separata (3)

- Tipicamente :





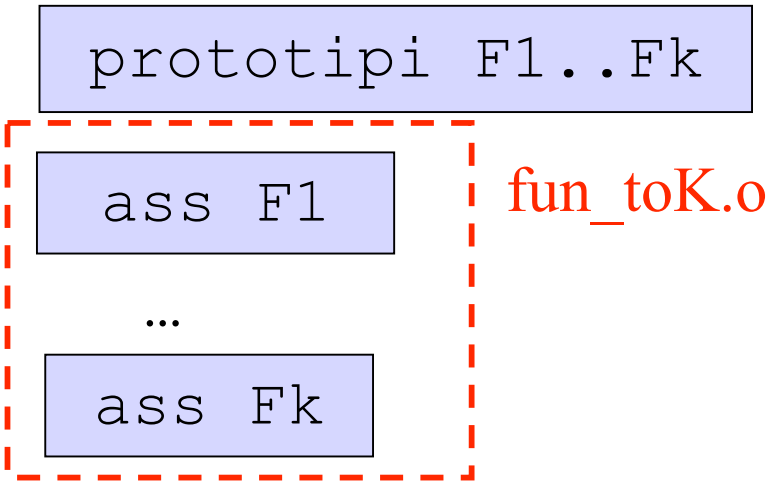
glob.h

```
#include "glob.h"
#include "fun_toK.h"
#include "fun_toN.h"
.....
```

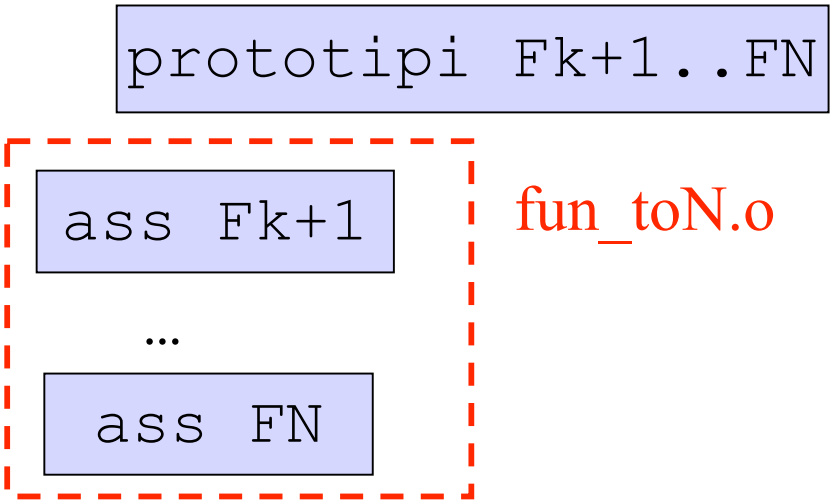
main.c

fun_toK.h

fun_1toN.h



fun_toK.o



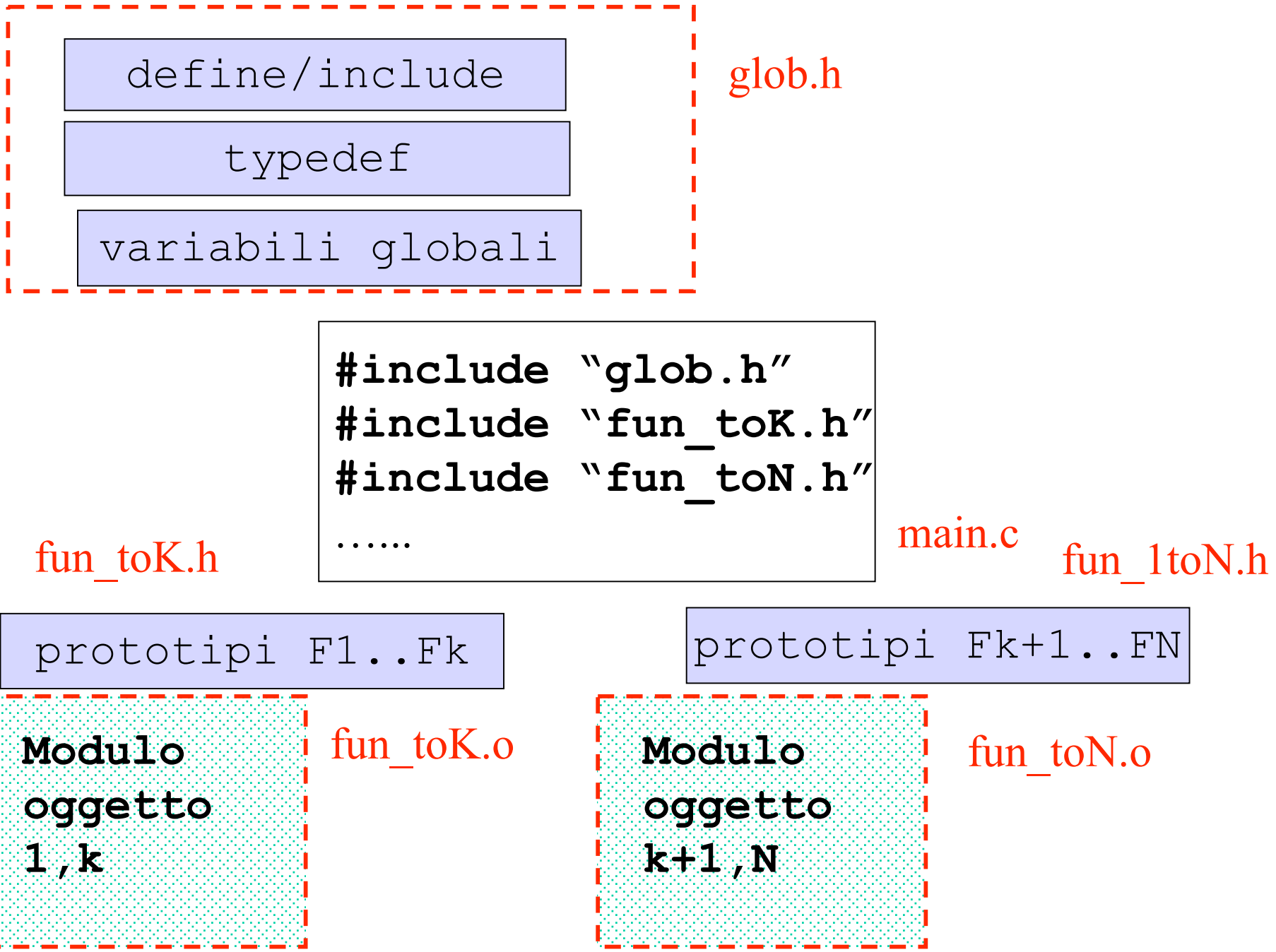
fun_toN.o

Compilazione separata (4)

- Per utilizzare delle funzioni definite in file diversi basta
  - (1) specificare il prototipo (dichiarazione) prima dell'uso
    - tipicamente includendo un file che lo contiene
  - (2) avere a disposizione una versione precompilata delle funzioni stesse (il modulo oggetto relativo al file che le contiene)
- Perché questo basta ?

Compilazione separata (5)

- Perché basta ?
  - Il prototipo della funzione permette al compilatore di fare i controlli di tipo
    - ogni funzione di cui non è noto il prototipo viene considerata **`void -> int`**
    - questo genera errori strani ma non fa fallire la compilazione (provate a non includere stdio.h…)
  - Dal modulo oggetto si può generare correttamente l'eseguibile finale del programma che utilizza le funzioni definite separatamente senza ricompilarle da capo (cioè dal sorgente complessivo)

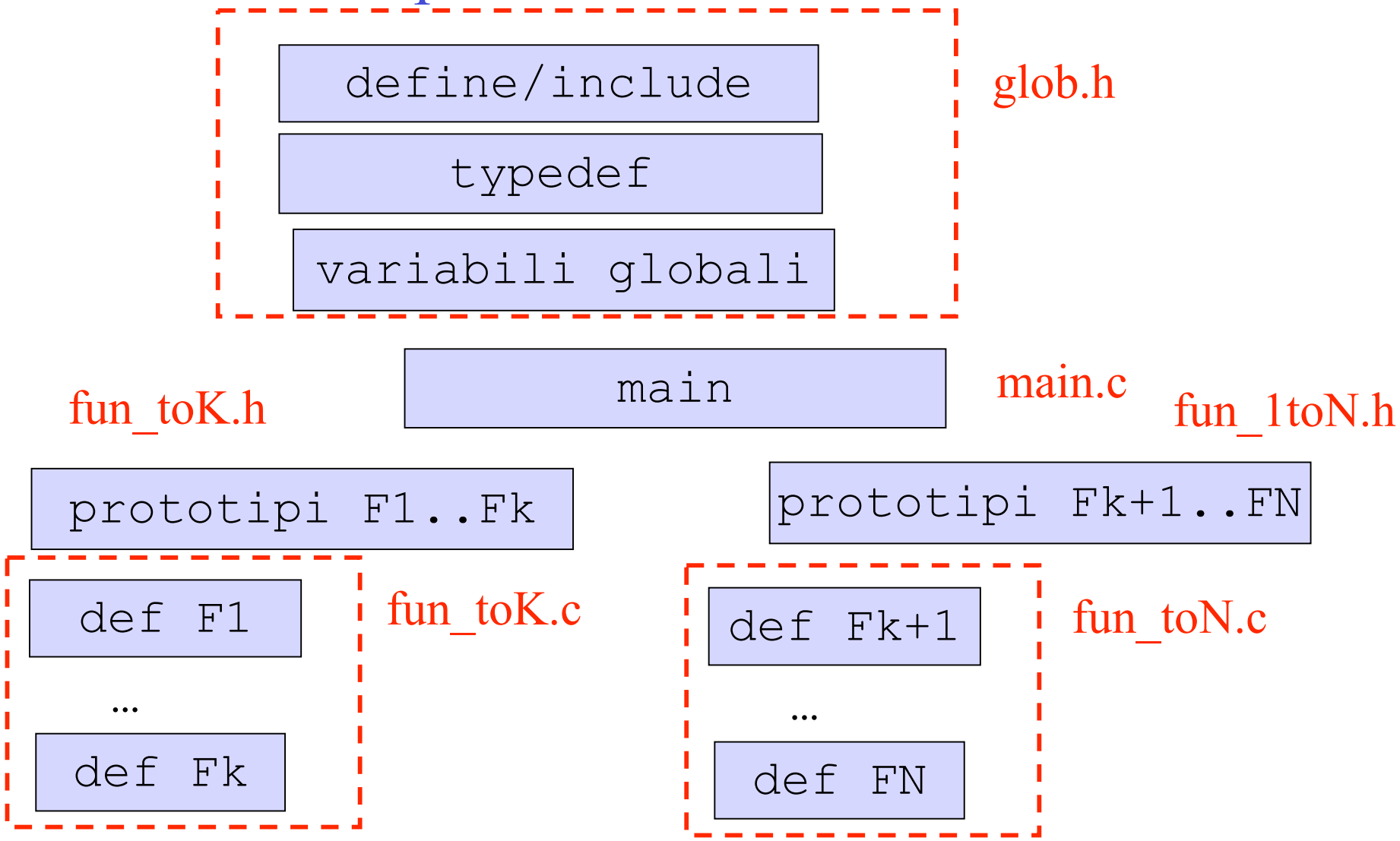

define/include
typedef
variabili globali
glob.h
#include "glob.h"
#include "fun_toK.h"
#include "fun_toN.h"
.....
main.c
fun_toK.h
fun_1toN.h
prototipi F1..Fk
prototipi Fk+1..FN
Modulo oggetto 1,k
fun_toK.o
Modulo oggetto k+1,N
fun_toN.o

Compilazione separata

Seconda parte

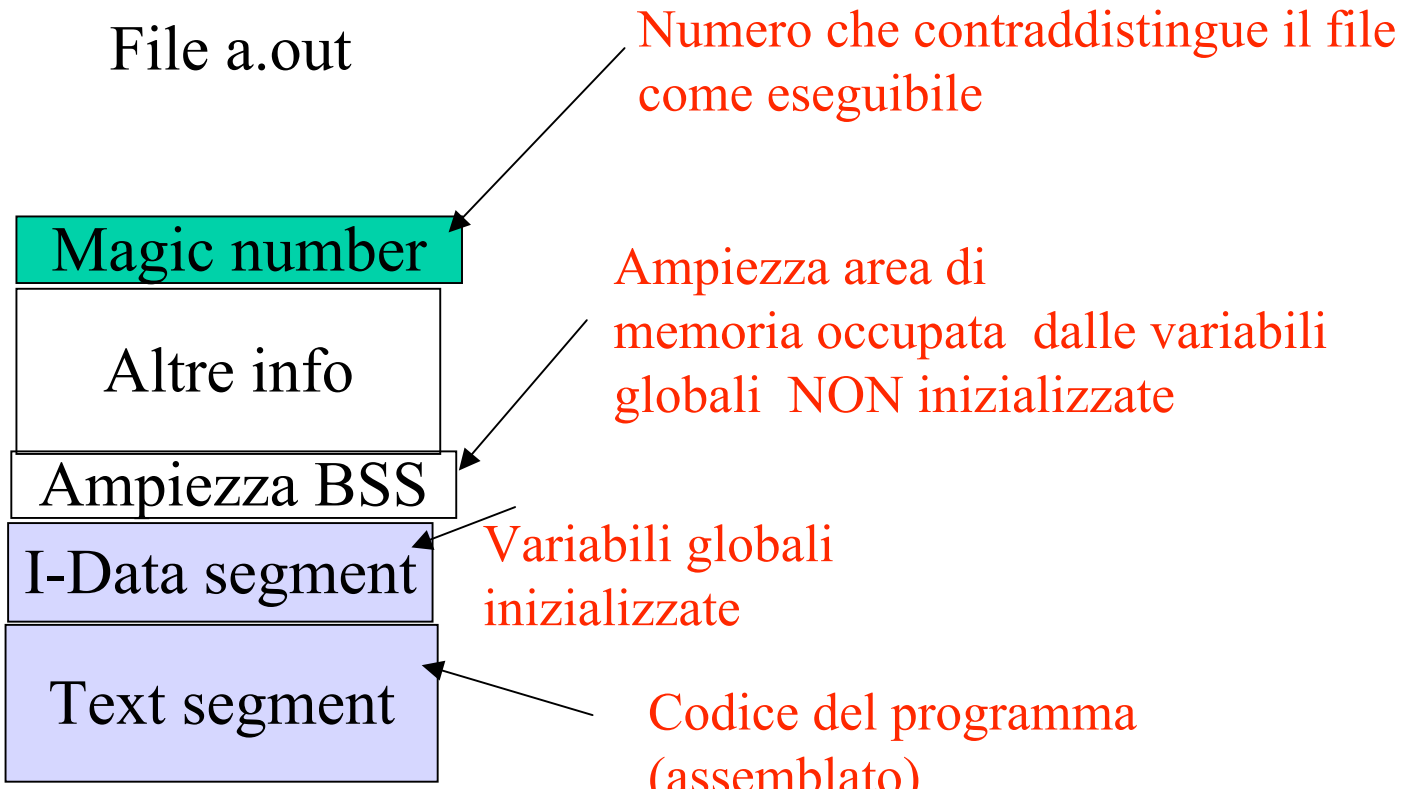
Compilazione separata (5.1)

- Partiamo da più file di testo



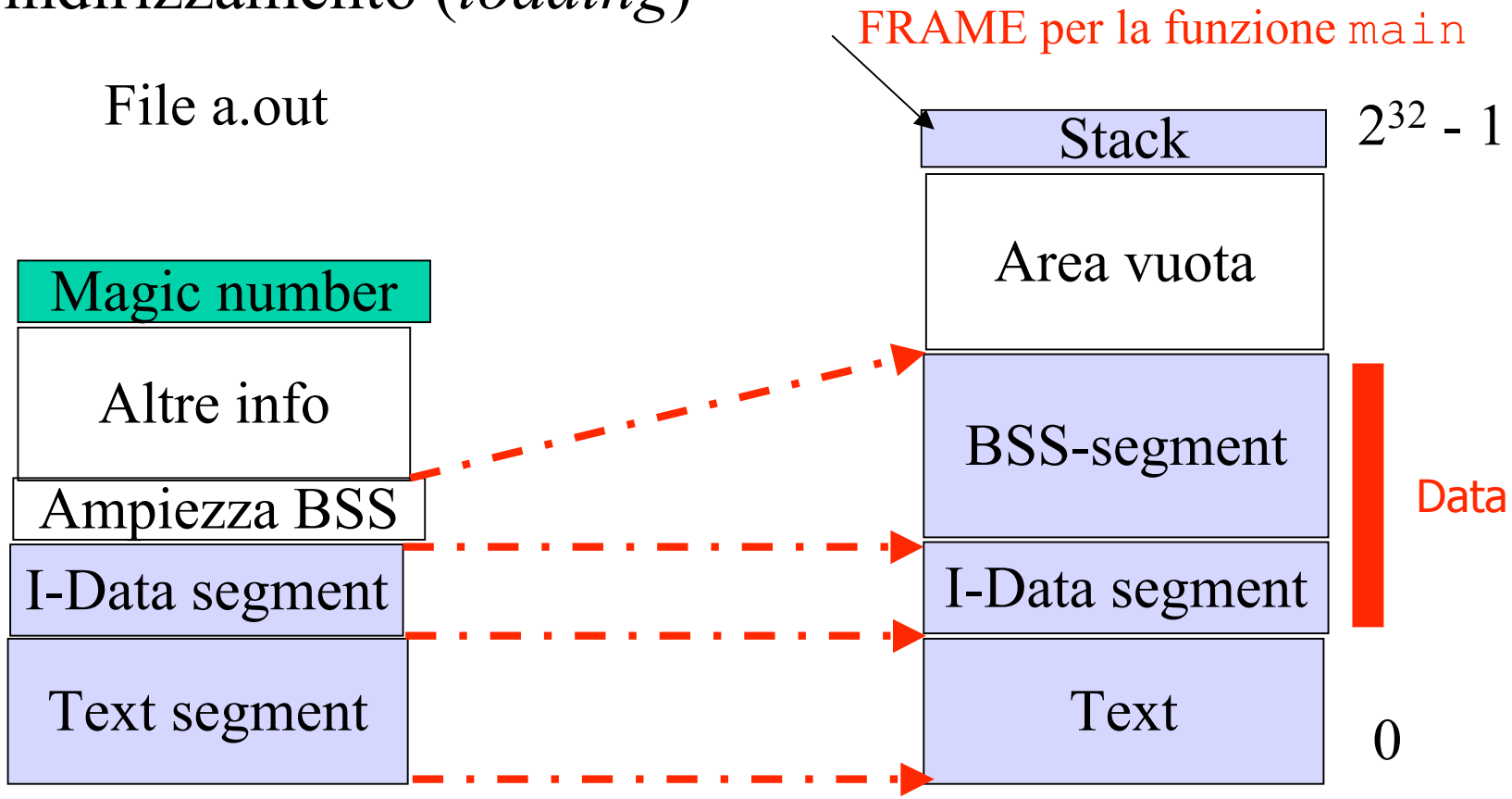
Compilazione separata (5.2)

- Vogliamo ottenere un unico eseguibile
  - Formato di un eseguibile ELF

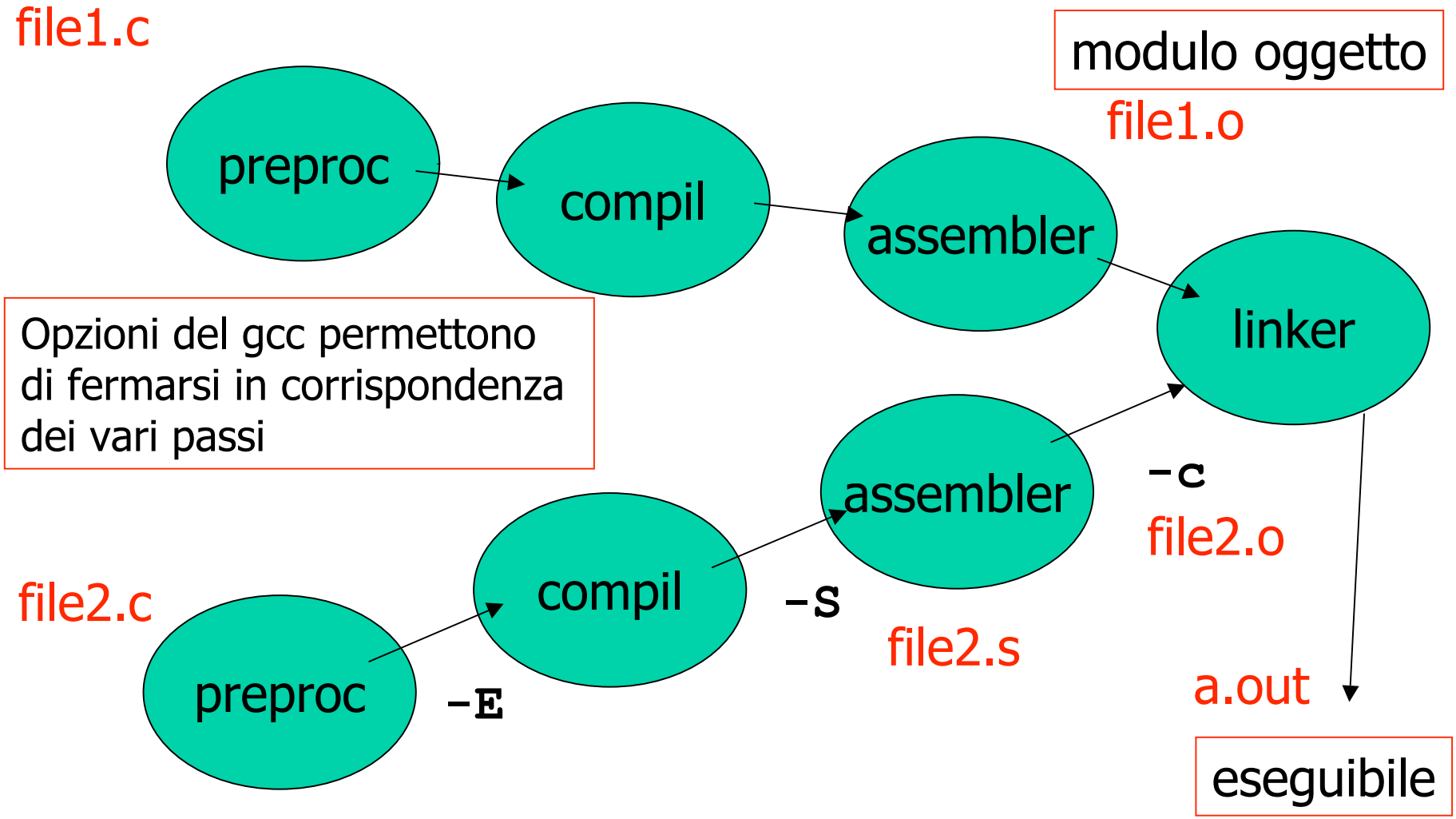


Compilazione separata (5.3)

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



Compilazione separata (6.0)

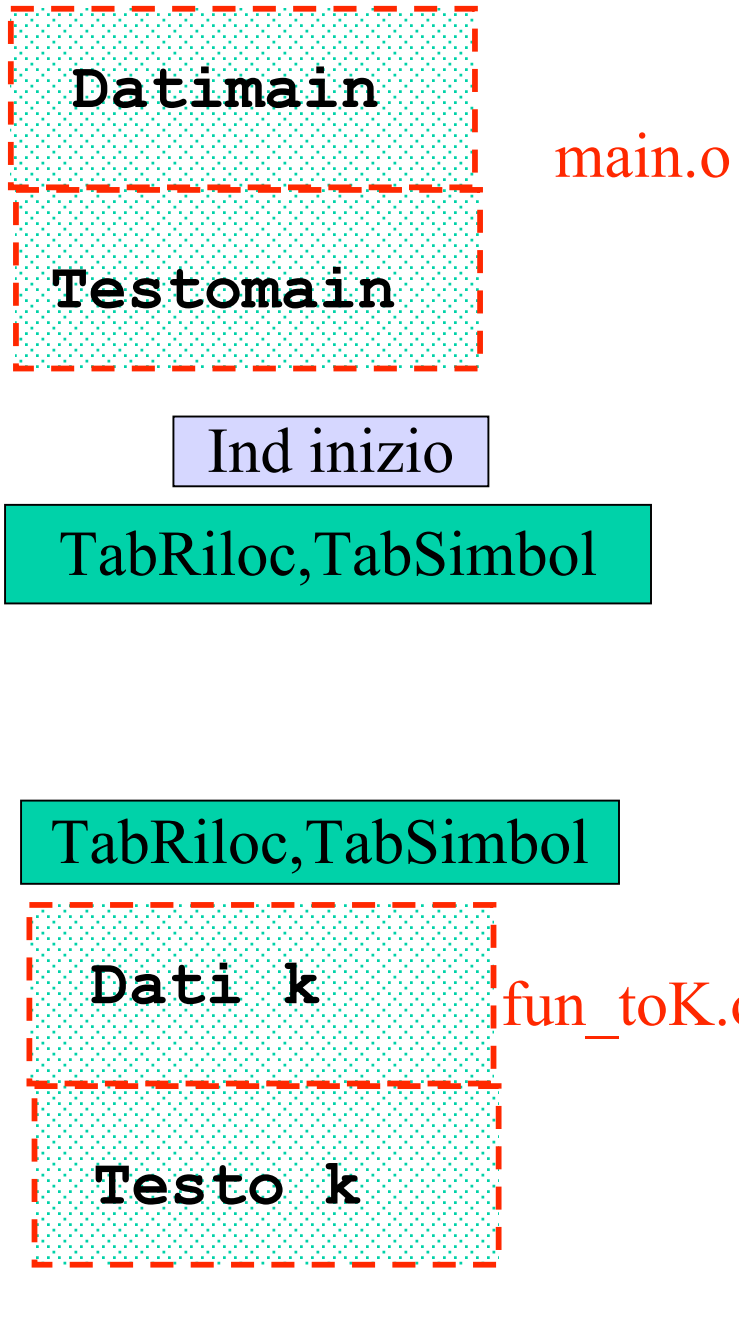


Compilazione separata (6)

- Come si crea il modulo oggetto?
  - **`gcc -c file.c`** produce un file **`file.o`** che contiene il modulo oggetto di **`file.c`**
  - Il formato dell'oggetto dipende dal sistema operativo
  - Che informazioni contiene l'oggetto ?
    - L'assemblato del sorgente **testo e dati** (si assume di partire dall'indirizzo 0)
    - La **tabella di rilocazione**
    - La **tabella dei simboli** (esportati ed esterni)

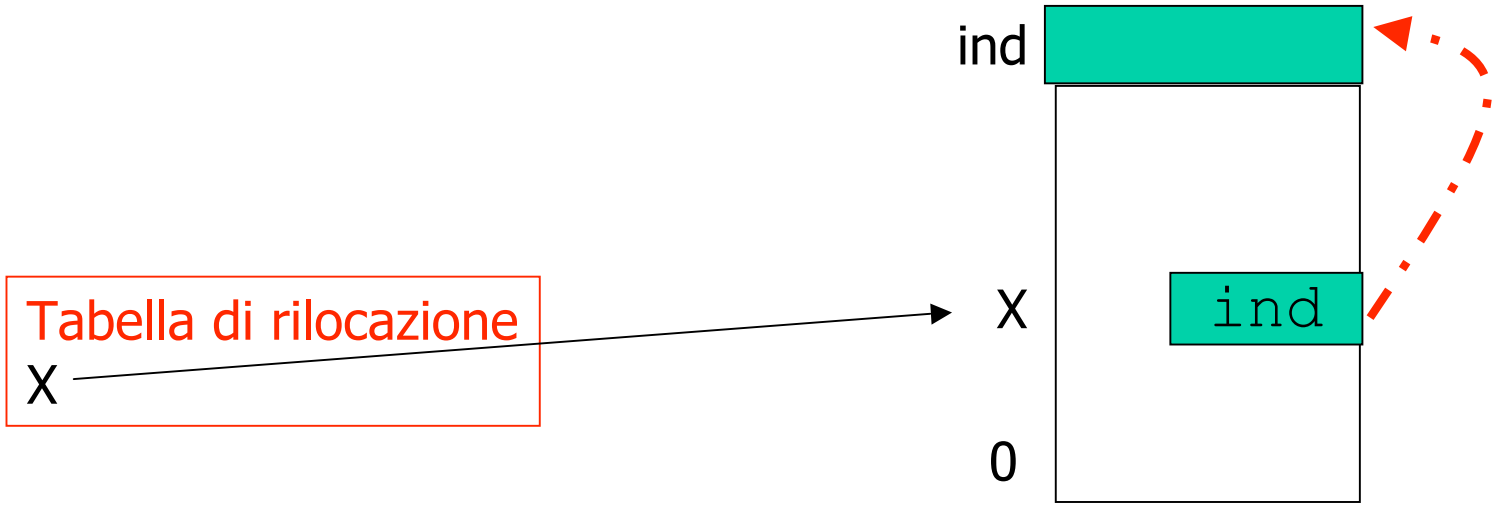
Compilazione separata (7)

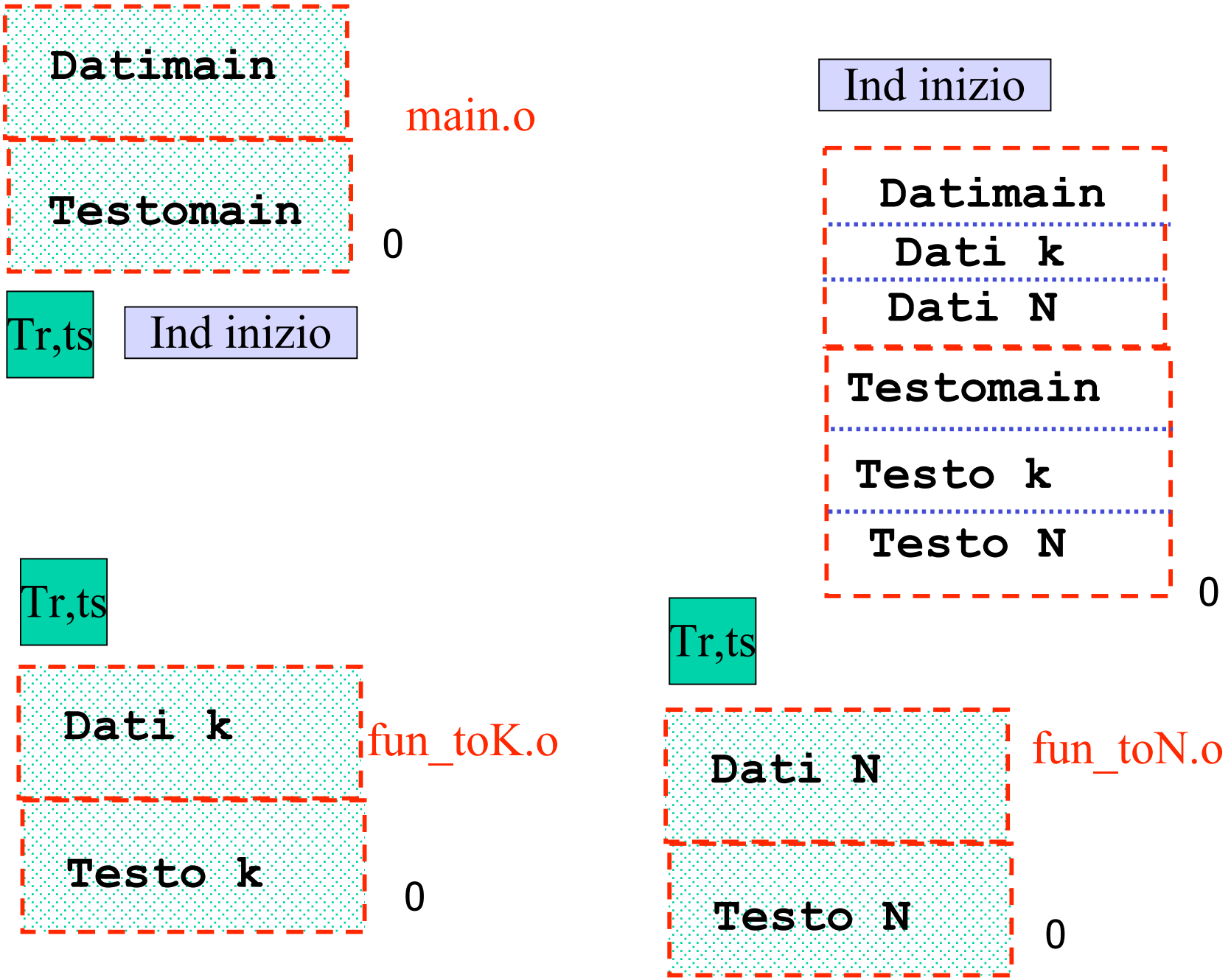
- Tabella di rilocazione
  - identifica le parti del testo che riferiscono indirizzi assoluti di memoria
    - es. JMP assoluti, riferimenti assoluti all'area dati globali (LOAD, STORE…)
  - questi indirizzi devono essere rilocati nell'eseguibile finale a seconda della posizione della prima istruzione del testo (`offset`)
  - all'indirizzo contenuto nell'istruzione ad ogni indirizzo rilocabile va aggiunto `offset`


Datimain
Testomain
main.o
Ind inizio
TabRiloc,TabSimbol
TabRiloc,TabSimbol
Dati k
Testo k
fun_toK.o
Situazione iniziale eseguibile
?
?
TabRiloc,TabSimbol
Dati N
Testo N
fun_toN.o

Compilazione separata (8)

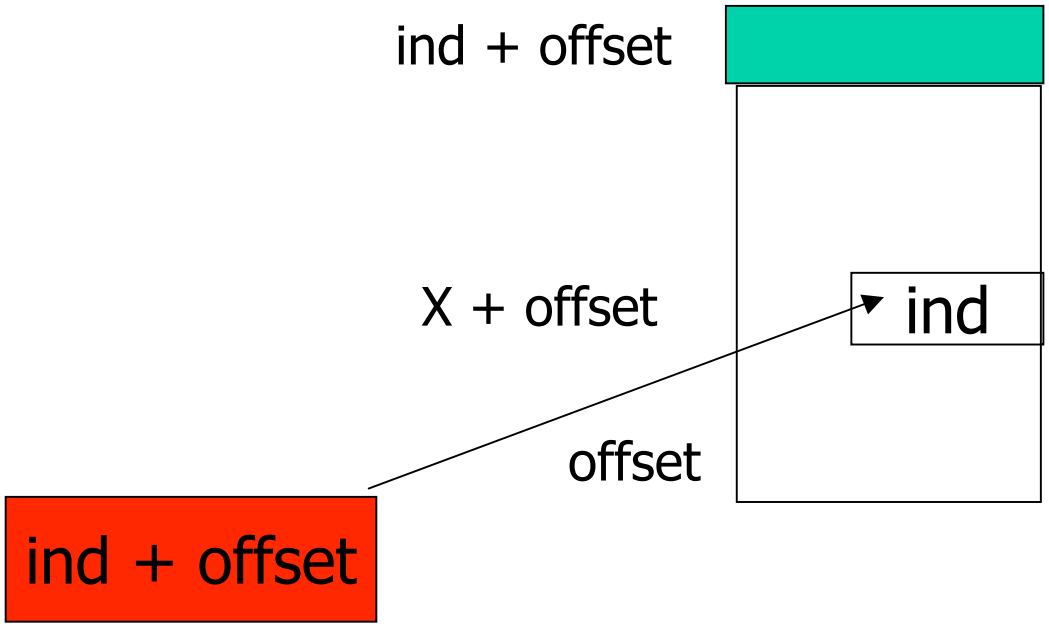
- Tabella di rilocazione (cont.)
  - il codice pre-compilato è formato da testo e dati binari
  - l'assemblatore assume che l'indirizzo iniziale sia 0




Datimain
main.o
Testomain
0
Tr,ts
Ind inizio
Ind inizio
Datimain
Dati k
Dati N
Testomain
Testo k
Testo N
0
Tr,ts
Dati k
fun_toK.o
Testo k
0
Tr,ts
Dati N
fun_toN.o
Testo N
0

Compilazione separata (9)

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



Compilazione separata (10)

- 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 (tipo `printf()`)

Compilazione separata (11)

- 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

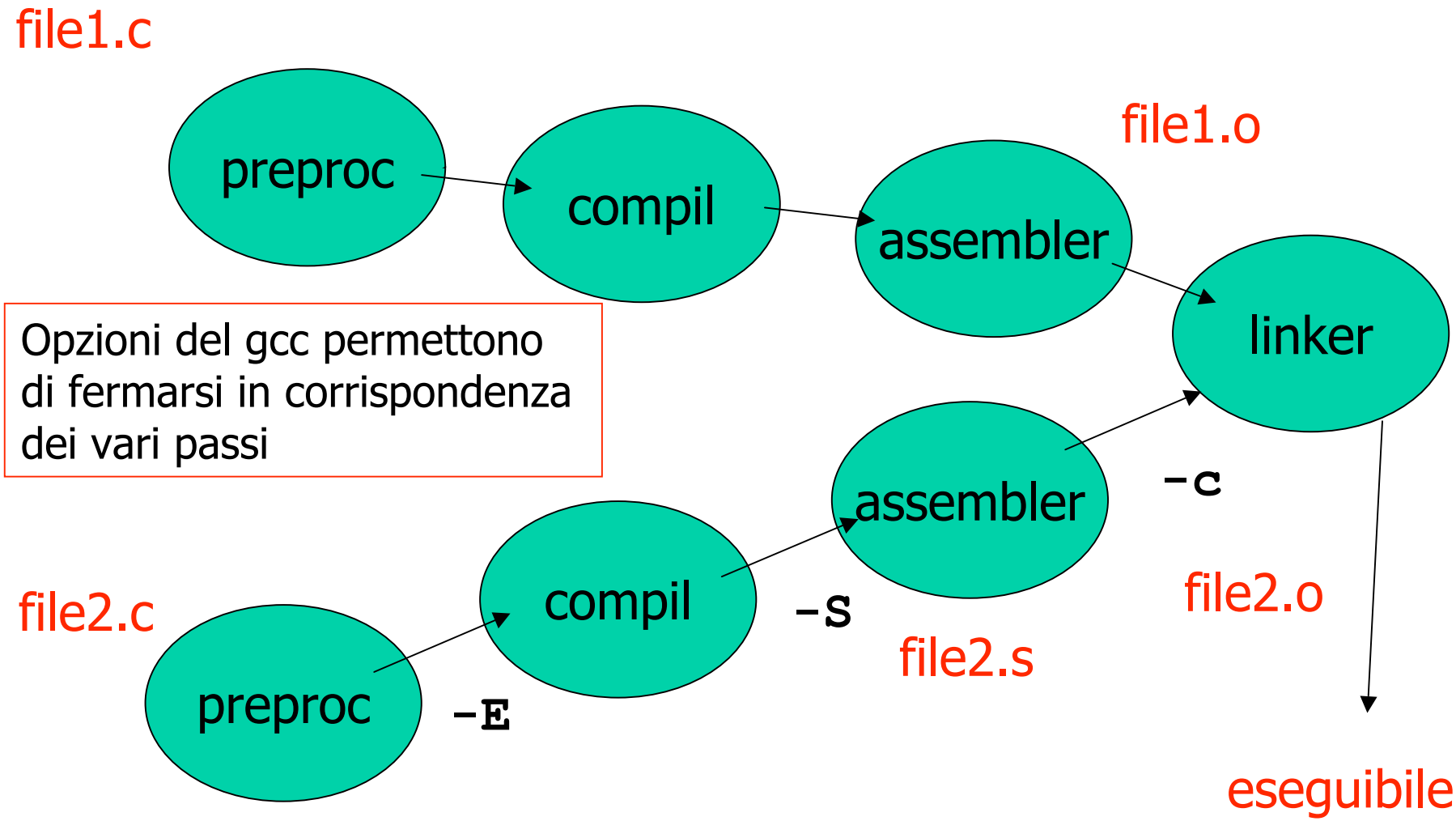
Compilazione separata (12)

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

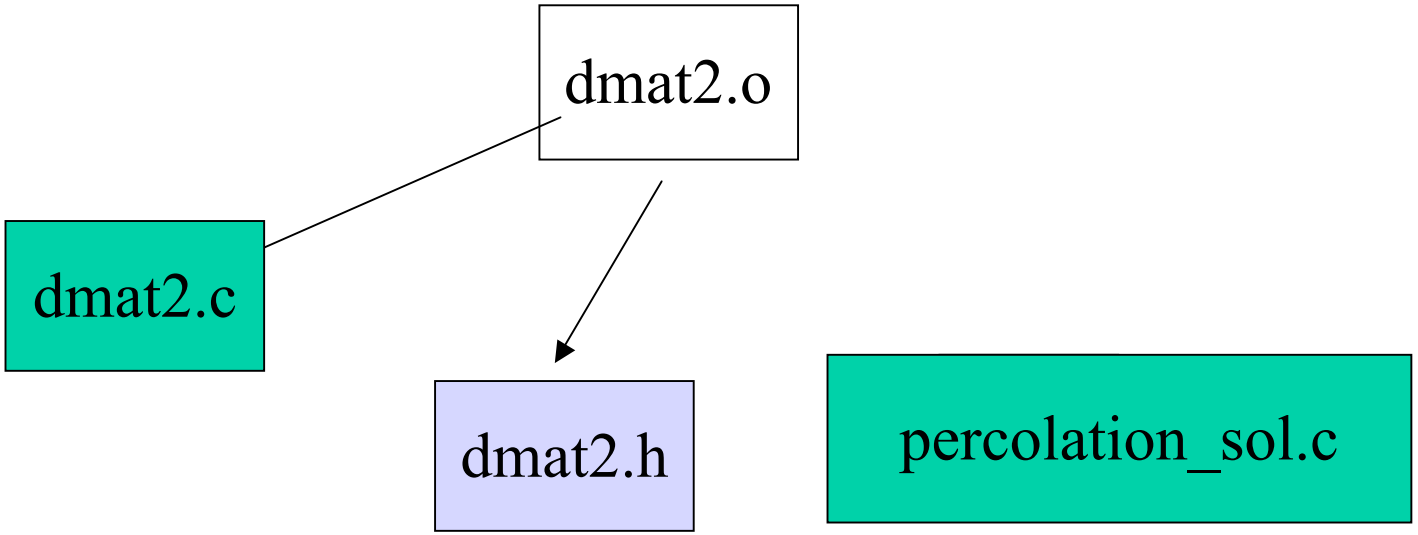
Compilazione separata (12.1)

- Il *linker* si occupa di risolvere i simboli (cont.)
  - Se il linker trova il simbolo esterno
    - eventualmente 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 ...

Compilazione separata (13)



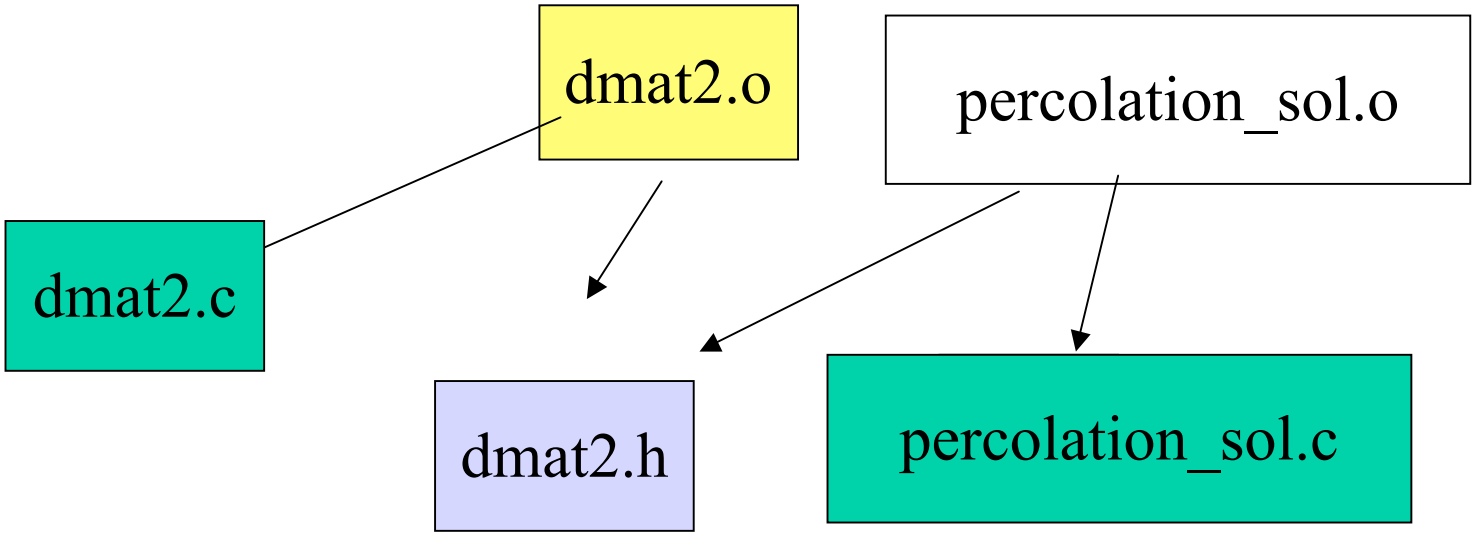
Esempio: percolation ...



Come costruire l'eseguibile (1)

```
$gcc -Wall -pedantic -c dmat2.c
      --crea dmat2.o
```

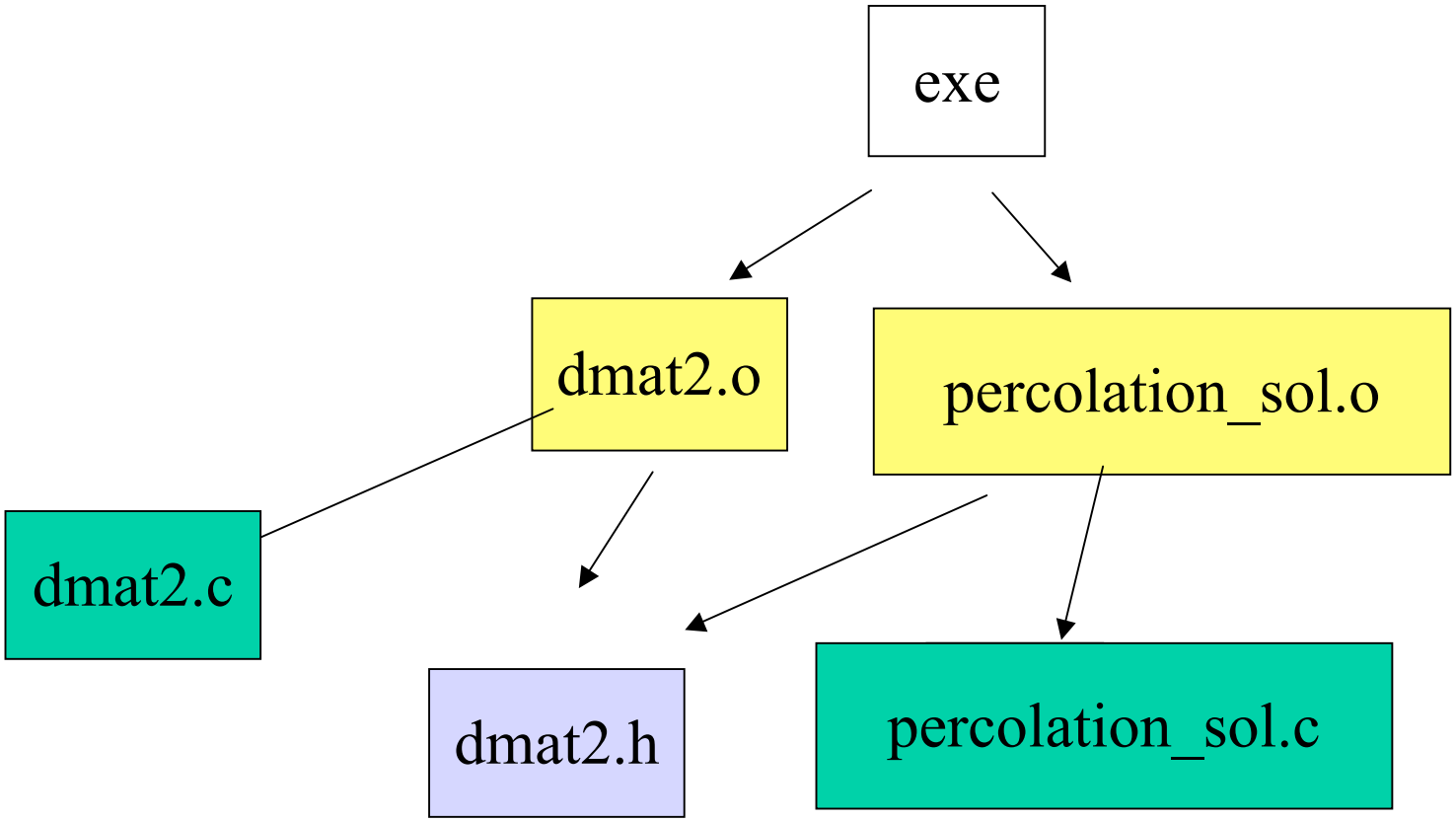
Esempio: percolation … (2)



Come costruire l'eseguibile (2)

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

Esempio: percolation … (3)



Come costruire l'eseguibile (3)

```
$gcc dmat2.o percolation_sol.o -o exe
  --crea l'eseguibile 'exe'
```

Esempio: percolation … (4)

```
$gcc -Wall -pedantic -c dmat2.c
  --crea dmat2.o
$gcc -Wall -pedantic -c percolation_sol.c
  --crea percolation_sol.o
$gcc dmat2.o percolation_sol.o -o exe
 --crea l'eseguibile 'exe'
```

- se modifico **`dmat2.c`** devo rieseguire (1) e (3)
- se modifico **`dmat2.h`** devo rifare tutto

Esempio: percolation … (5)

```
$gcc -M dmat2.c
   --fa vedere le dipendenze da tutti i
    file anche dagli header standard delle
    librerie
dmat2.o : dmat2.c /usr/include/stdio.h \
          /usr/include/sys/types.h \
          … … … … …
```

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

Come visualizzare i moduli oggetto

- Comando **`nm options file.o`** fornisce tutti i simboli definiti in **`file.o`**
  - **`$nm -g dmat2.o`**

  fornisce solo i simboli esportati
- Comandi **`objdump`** e **`readelf`** permettono di leggere le varie sezioni dell'eseguibile
  - **`$objdump -d dmat2.o`**

  fornisce il testo disassemblato
  - **`-r`** tabelle di rilocazione
  - **`-t`** symbol table
- Vedere anche **`info binutils`** da emacs