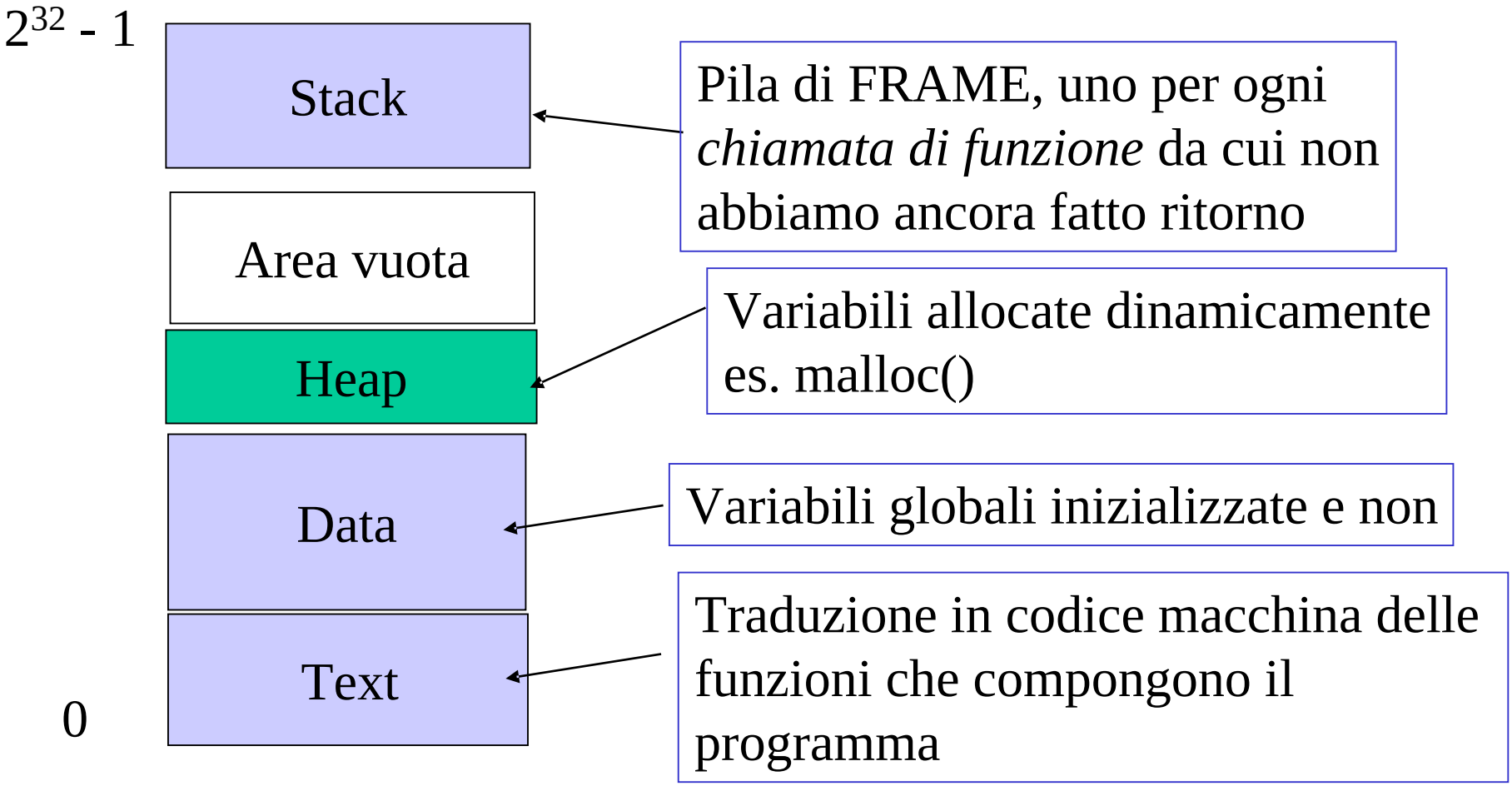
Preprocessing, compilazione ed esecuzione

Utilizzando strumenti GNU...

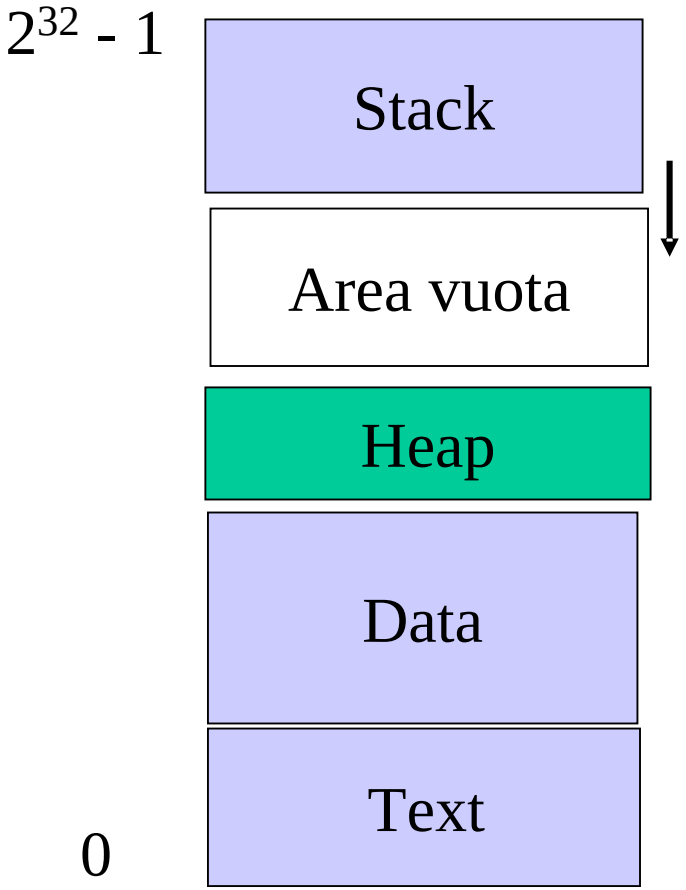
Spazio di indirizzamento

- Come vede la memoria un programma C in esecuzione



Spazio di indirizzamento (2)

- Lo stack

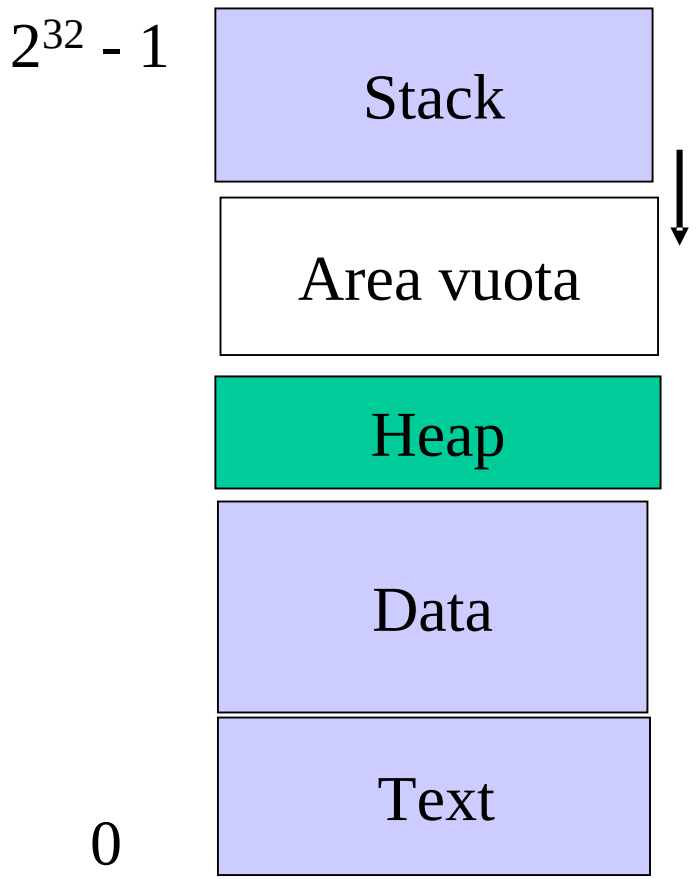


Direzione di
crescita dello stack

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

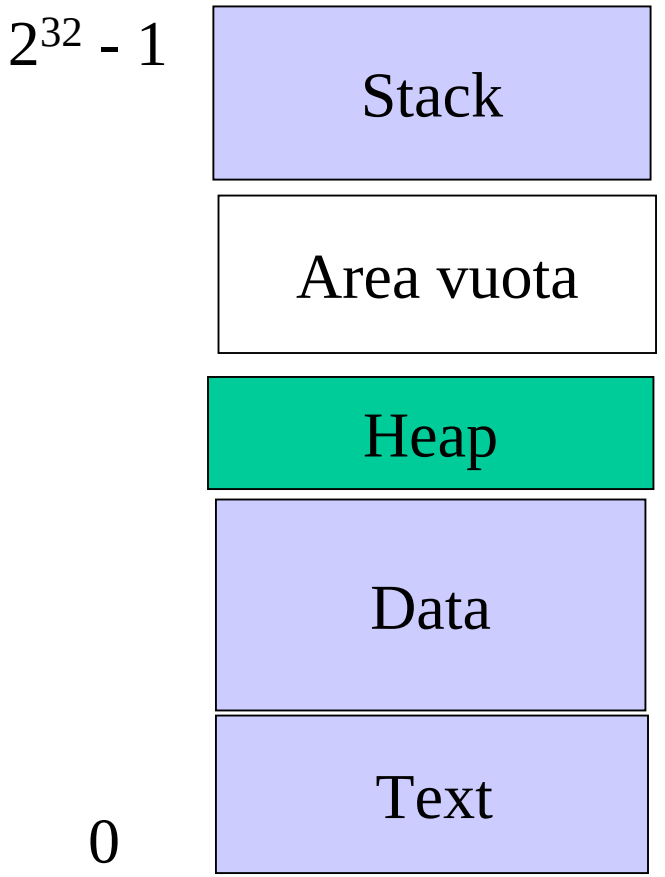


All'inizio dell'esecuzione lo Stack contiene solo il FRAME per la funzione `main`

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

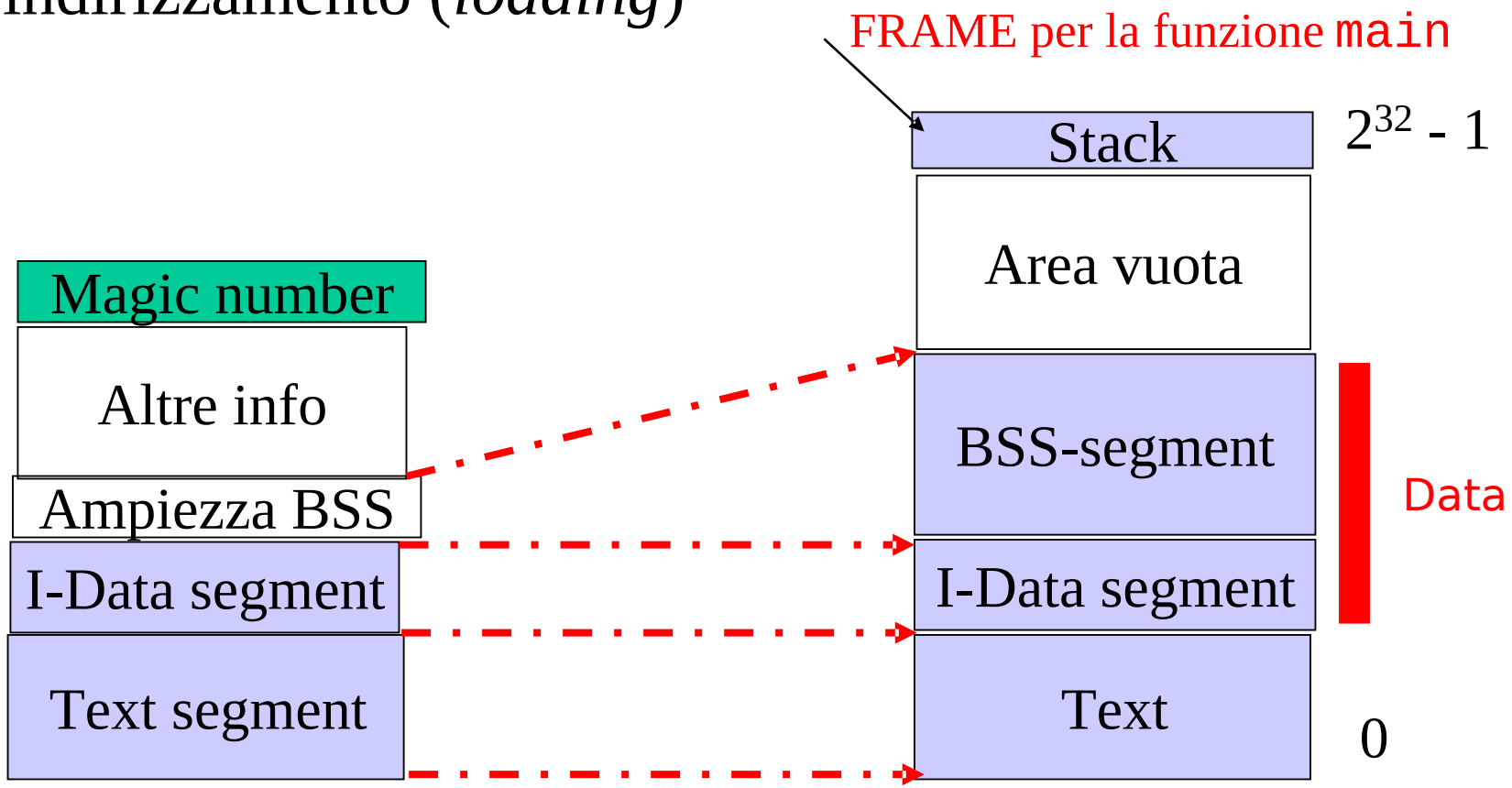
- La compilazione produce un file eseguibile
- Il formato di un eseguibile dipende dal sistema operativo
- In Linux un eseguibile ha il formato ELF (*Executable and Linking Format*)
  - eseguibili e moduli oggetto hanno lo stesso formato
  - assembler + tabelle varie
  - tabelle eliminabili con il comando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

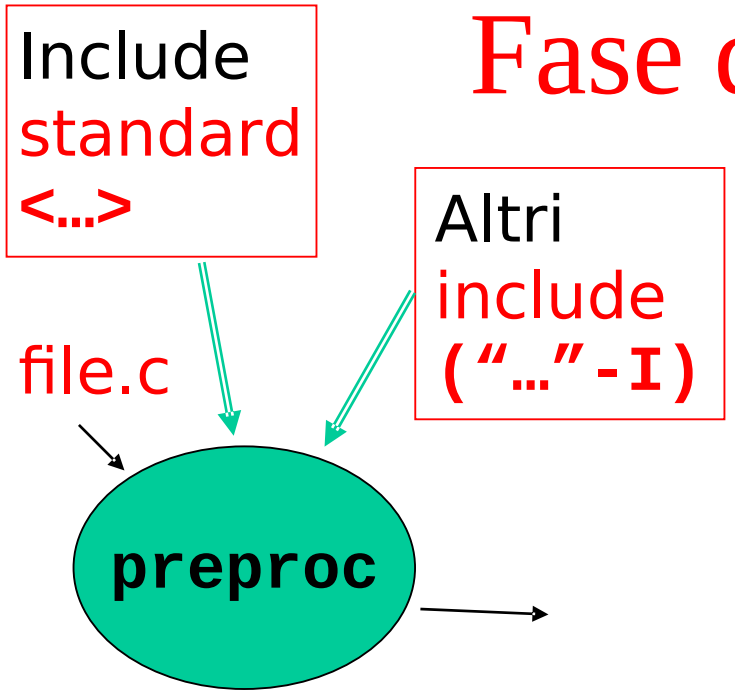


C: dal sorgente all'eseguibile

- 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***

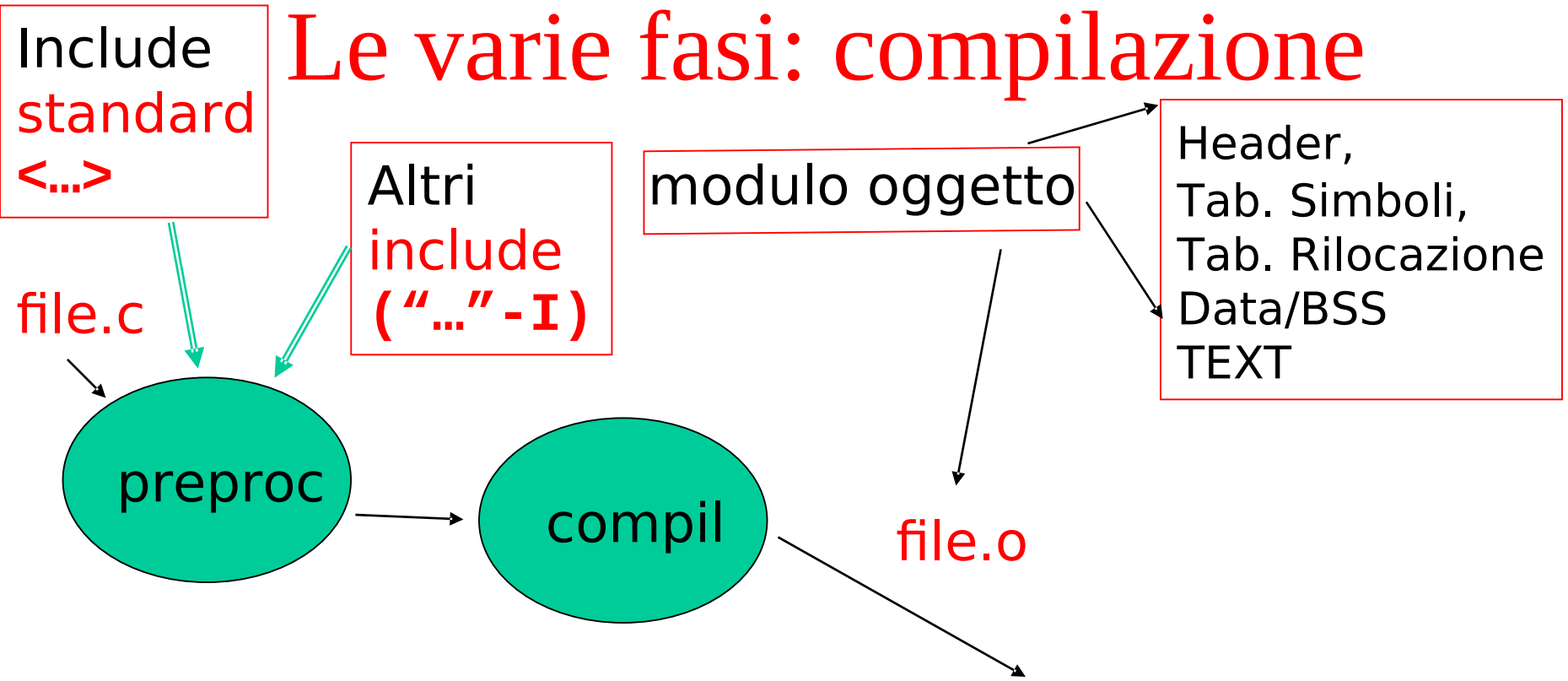
```
#include <stdio.h>
#define N 10
int max = 0;
int main (void){
  int i, tmp;
  printf("Inserisci %d interi positivi\n",N);
  for (i = 0; i < N; i++) {
     scanf("%d", &tmp);
     max = (max > tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

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)
```

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;
}
```

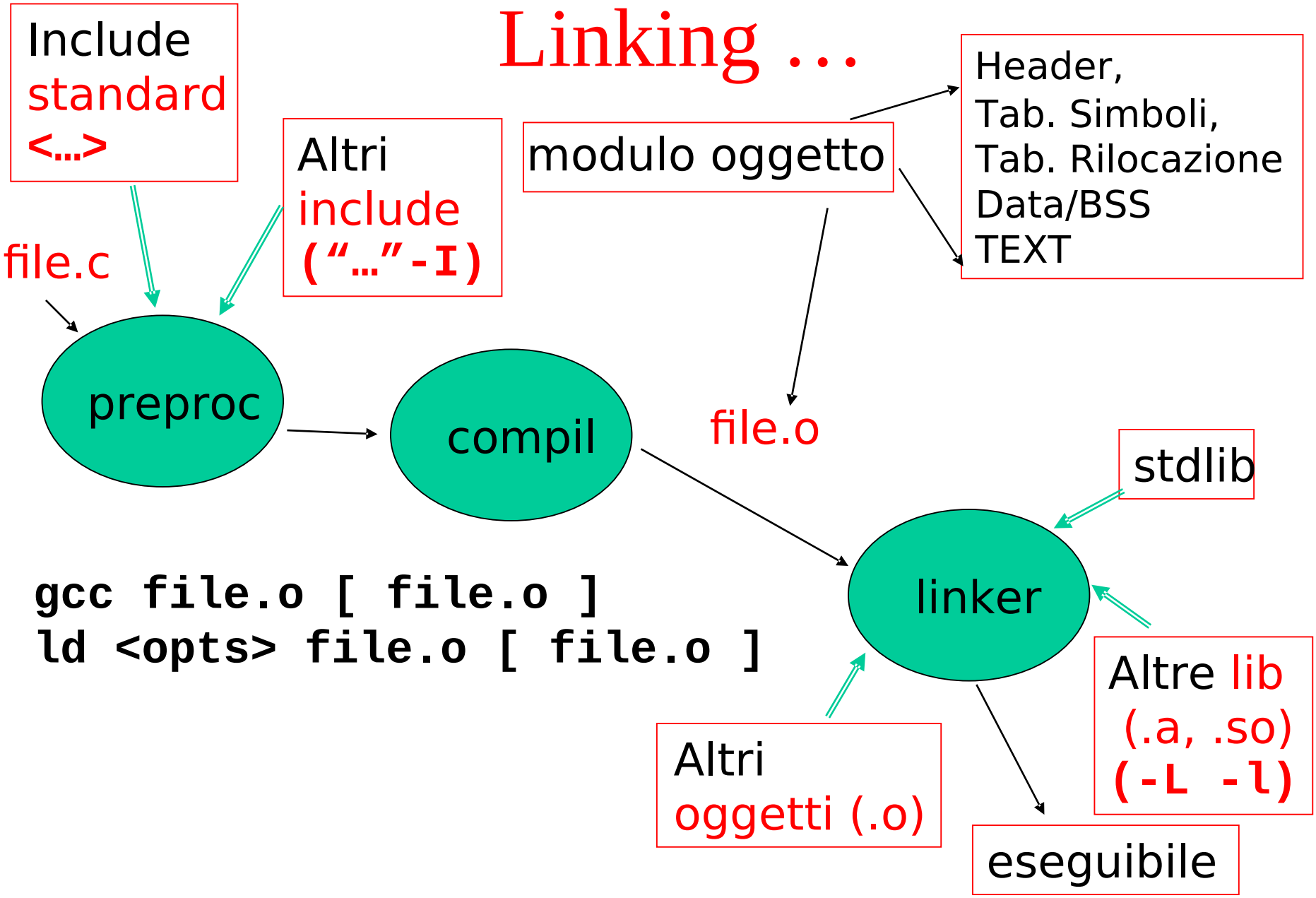
Compilazione : un esempio (6)

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;
}
```

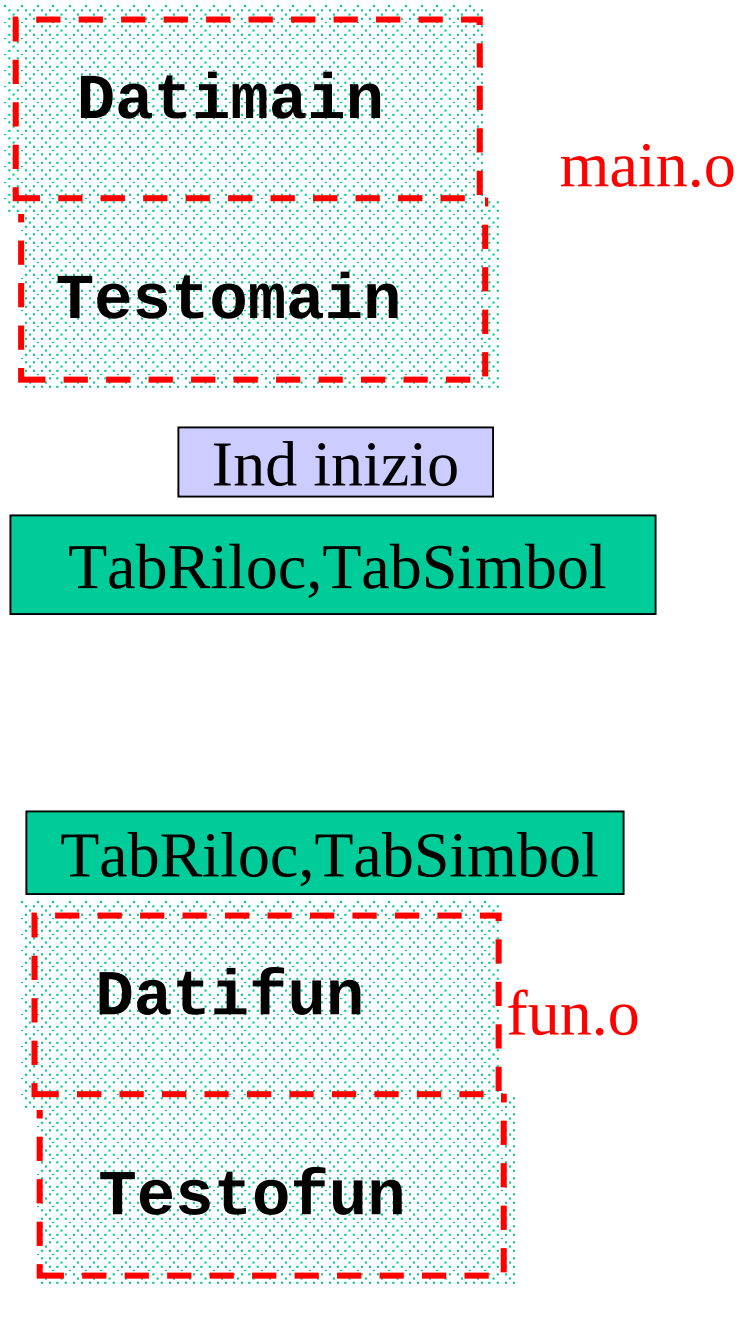
Linking …

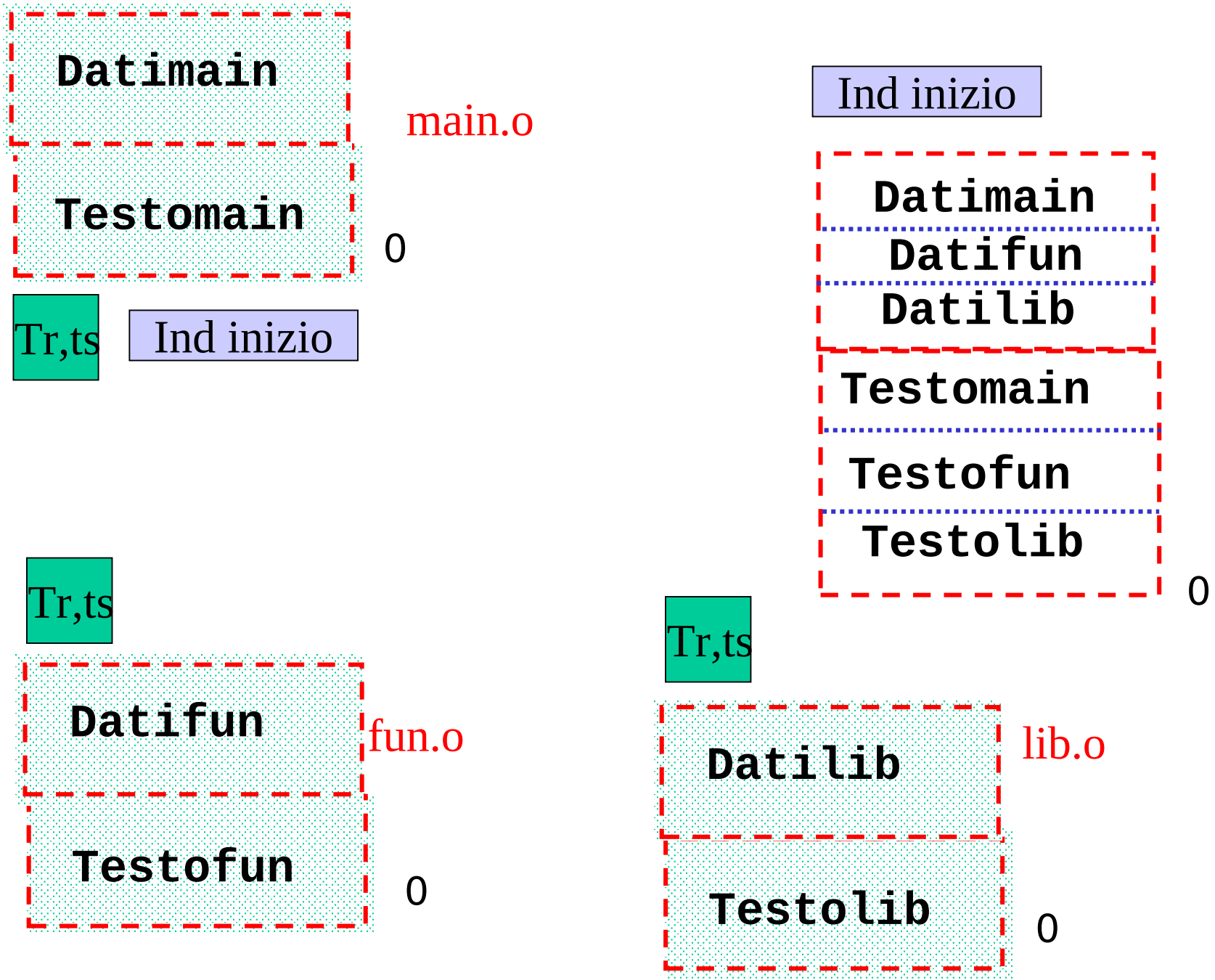



```
gcc file.o [ file.o ]
ld <opts> file.o [ file.o ]
```

Linking: esempio statico

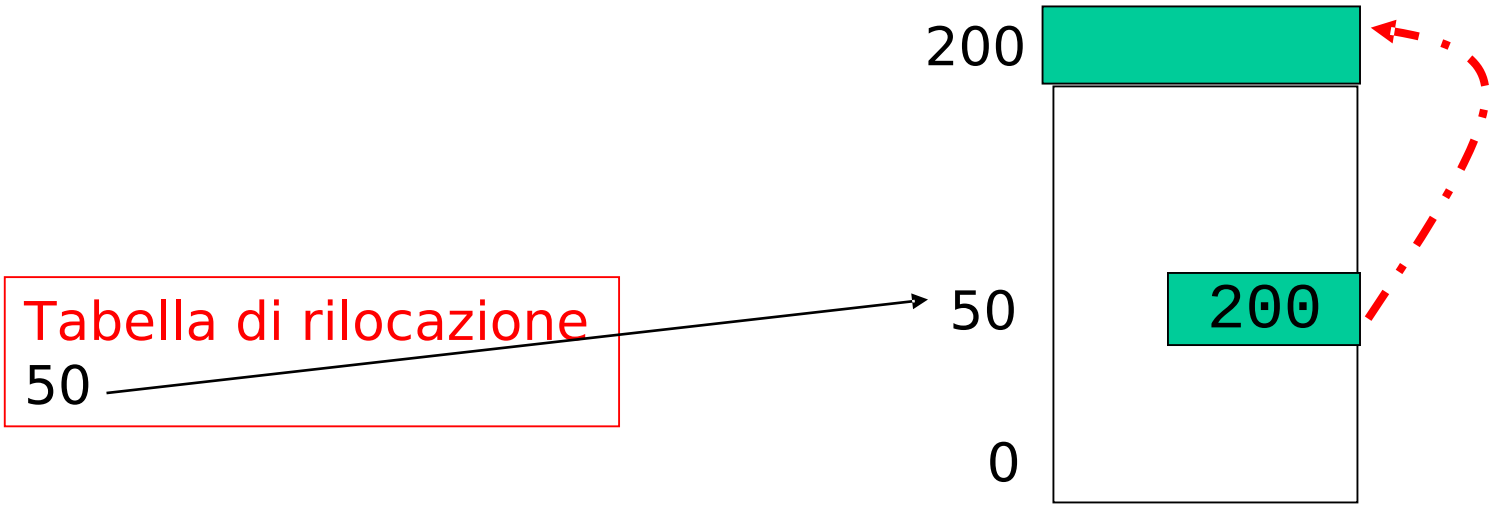
- Si collegano assieme più moduli oggetto
  - **`file.o`** oppure librerie di oggetti **`libfile.a`**
  - ...per creare un file eseguibile
- Ogni modulo oggetto contiene
  - L'assemblato del sorgente **testo e dati** (si assume di partire dall'indirizzo 0)
  - La **tabella di rilocazione**
  - La **tabella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


Datimain
Testomain
main.o
Ind inizio
TabRiloc,TabSimbol
TabRiloc,TabSimbol
Datifun
Testofun
fun.o
Situazione iniziale eseguibile
?
?
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
Testofun
fun.o
0
Tr,ts
Datilib
Testolib
lib.o
0

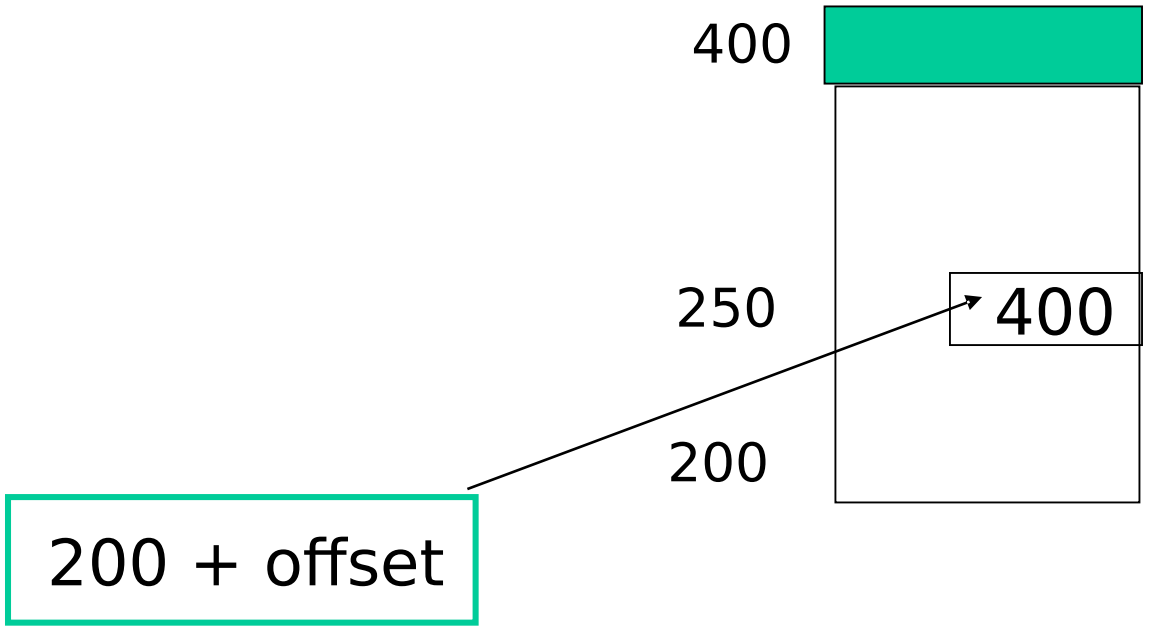
Linking: esempio statico (3)

- 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)

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`)

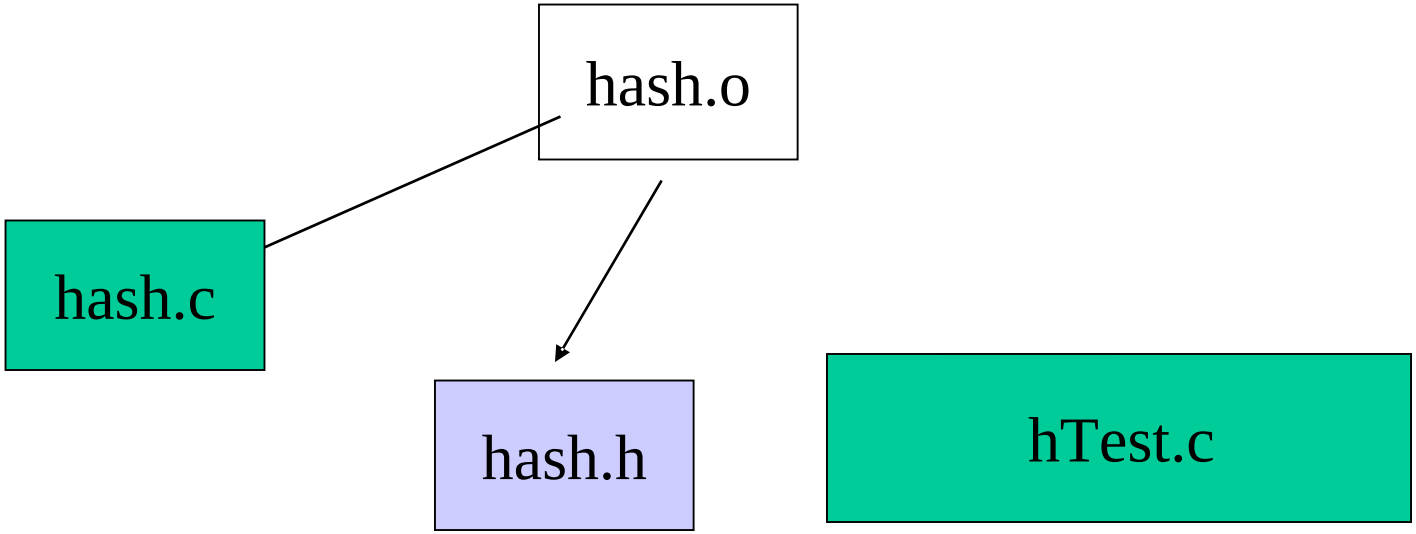
Linking: esempio statico (8)

- 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 ...

Esempio

Compilare e linkare correttamente un piccolo progetto

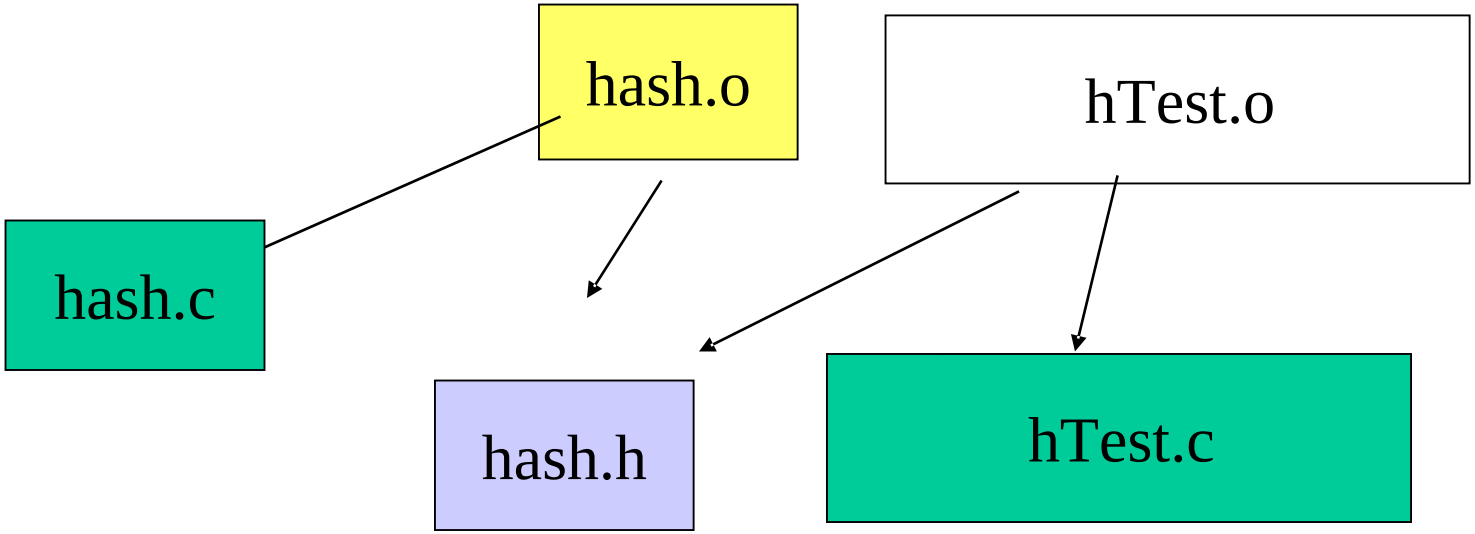
Esempio: tabella hash



Passo (1):

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

Esempio: hash … (2)

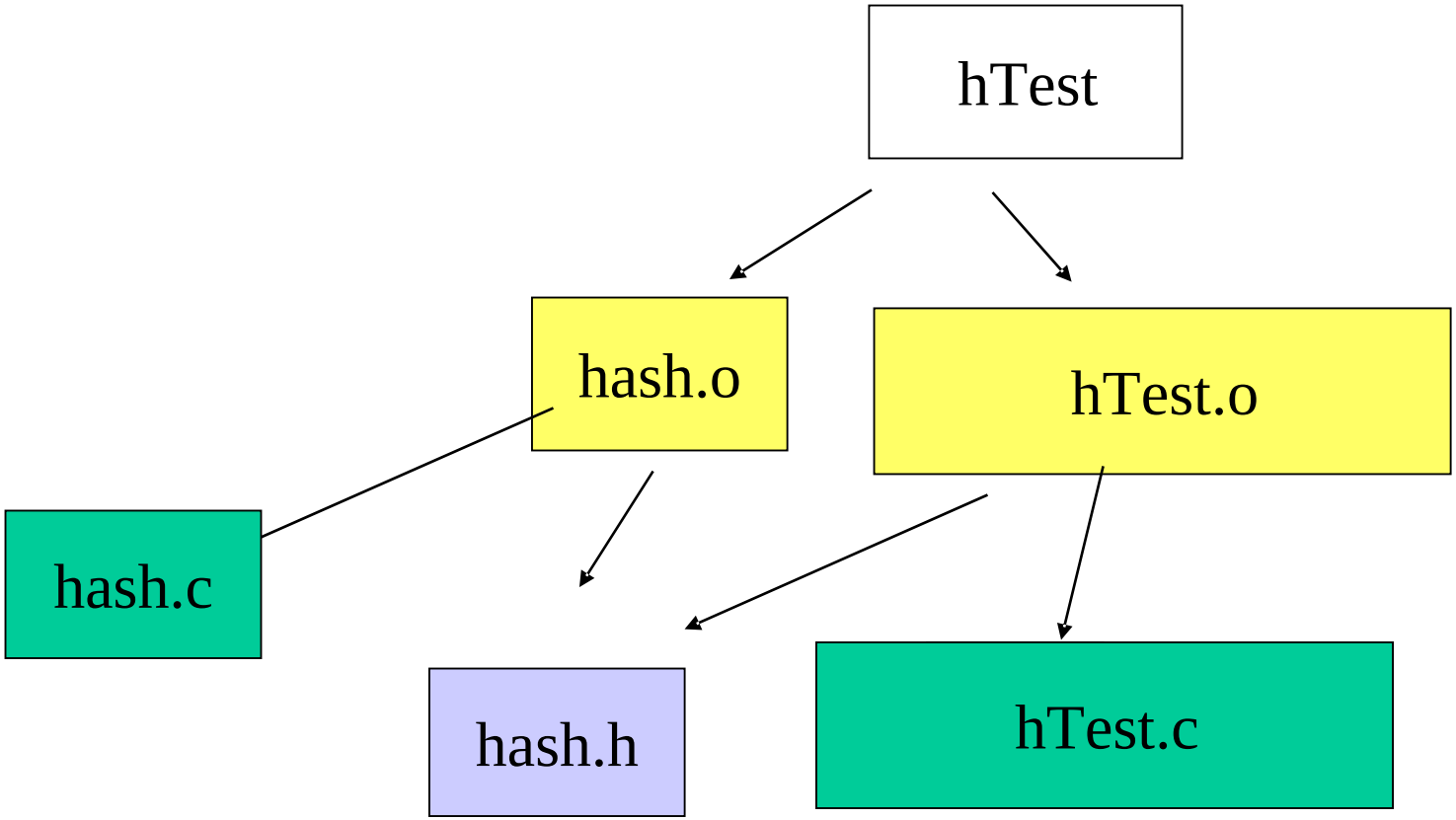


Come costruire l'eseguibile: passo (2):

```
$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 ….